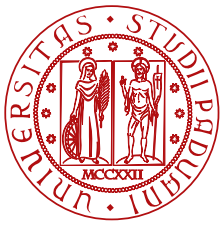


UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

MASTER'S DEGREE IN BIO-ENGINEERING

Orange Virtual Assistant: Investigating Large Language Models' Ability to Understand and Construct Data Mining Workflows

Supervisor
Prof. Baruzzo Giacomo

Student
Tiveron Alessandro

Co-Supervisors
Prof. Zupan Blaž
Prof. Di Camillo Barbara

ACADEMIC YEAR 2024-2025

Graduation date 26/03/2025

Abstract

This thesis looks into how important workflow processes can be automated by Large Language Models (LLMs) like GPT-4 to improve user engagement with Orange, a data mining application. The research specifically focuses on three primary goals: naming workflows, describing workflow functionality, and recommending new widgets to users based on partially completed workflows. Using methods such as prompt engineering and workflow similarity analysis, this study investigates LLMs' ability to comprehend and navigate enhance directed acyclic networks that serve as the foundation for Orange operations.

The findings show that workflow naming and description generating activities may be effectively completed by LLMs with high accuracy, especially when high quality examples are given. While advanced models produced encouraging results in the widget suggestion task, there are still certain issues, especially with limited existing workflow examples. The results show the strengths and weaknesses of existing LLMs in helping users in constructing and understanding data mining workflows. This thesis lays the groundwork for future advancements in workflow management and user assistance by giving insightful information about the incorporation of LLMs in a data mining setting.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The rapid improvement of Large Language Models (LLMs) like ChatGPT has changed the world in many ways, thanks to their efficiency and versatility. Their ability to "understand" and process text allows them to perform non-trivial tasks in a matter of seconds. This is not only useful for text, but can be generalized to many tasks by implementing specific prompts. Applications are starting to use LLM-powered virtual assistants to help new users learn how to use the app. This can also be done with Orange: a data mining application that allows its users to perform analysis on many types of datasets thanks to the implementation of workflows that allow users to perform data mining techniques through a high-level visual programming interface (Figure 1.1). In Orange, a workflow is a graph that contains working units, named widgets (vertices of the graph) that are connected with each other through input-output links (arcs of the graph). Given that there are 227 widgets available and that they can be connected to each other in many ways, these workflows are not trivial to understand for any inexperienced user, and the rise of the LLM-based virtual assistant represents a great opportunity for Orange to become more user-friendly.

As previously reported, this would not be the first time something like this has been implemented. In fact, Guan et al., 2023 [1] studied and created a way to make LLM perform application actions according to high-level user made requests. In this study, the LLM-augmented Autonomous Agent (LAA: the LLM-powered virtual assistant) was used to perform tasks in the target environment of Alipay mobile payments application. This LAA is based on modules that transform information from the app into natural language and vice versa, and modules that provide information about previous actions and possible next actions, thus allowing LLM to interact with the application and perform operations. The results observed on Alipay are very promising, as the LAA achieved a success rate of 70% on the selected tasks, thus opening up a

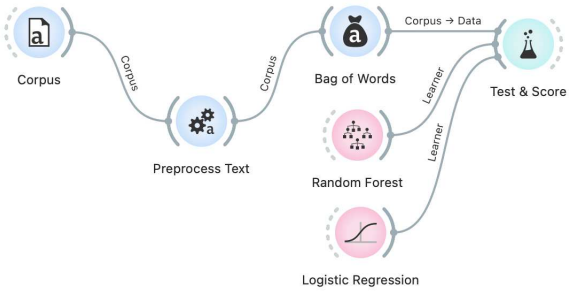


Figure 1.1: Example of an Orange Data Mining workflow that uses bag-of-words document representation for document classification purposes, and compares the accuracy of random forest and logistic regression models.

world of possibilities as LLMs become stronger and more efficient. The fact that Orange uses workflows for user-app interactions makes it very difficult to create an Orange-specific LLA, since they are effectively directed acyclic graphs. In fact, as reported in Fatemi et al., 2024 [2], the LLMs struggle to understand and perform operations on graphs, and do not perform much better than a random guess. However, accuracy is highly dependent on the model and type of prompt used to give the LLMs information about the graph. Wu et al., 2024 [3] on the other hand, performed a similar study, but differentiated the performance between normal graphs and directed graphs. On difficult tasks, all the models achieved an accuracy of 0%, except for the two most powerful LLMs (GPT-4 and GPT-4o) which achieved a slightly better, but still sub-optimal, accuracy of 50% and 20% for directed and undirected graphs, respectively. In general, the tasks performed on the directed graphs achieved better results than those performed on the undirected graphs, regardless of the model used.

The last two mentioned studies reported results obtained by analyzing and performing tasks on graphs, regardless of function of the nodes and meaning of the edges. In Orange workflows, we also have information about these two things, which can be beneficial to increase the functionality and efficiency of LLMs graph analysis. The following work aims at creating an LLM-powered virtual assistant for Orange, similar to the one created by Guan et al., 2023 [1], in order to help inexperienced users in using the application. To do this, we identified three different tasks that we want the virtual assistant to perform: invent a name for a given workflow, generate a description of the analysis performed by the workflow, and suggest which widget could come next in the workflow given the user's goal.

In this thesis, in the Background chapter we will first describe the Orange Data Mining application and its main uses, discuss the basics of graph theory, explain how chatbots work, by explaining what both LLMs and Natural Language Processing (NLP) are, and finally review state-of-the-art prompt engineering techniques and their application to LLMs such as GPT. Then,

in chapter 3, we will discuss the methods selected and used to do the following: extract the workflows from screenshots of the Orange data mining app, create a t-SNE representation of the workflows contained inside the screenshot dataset, and generate, through the utilization of LLMs, a given workflow's potential name, potential description, and suggestion of potential new widgets. An explanation of how the performance of these tasks is evaluated will also be provided. In chapter 4 we will discuss the results of the analysis described in the previous chapter and whether the goal of this thesis has been achieved. Finally, in the last chapter, we will summarize what has been done and draw conclusions. Additionally, at the end of the thesis, there will be an Appendix to showcase the Python library created and used to perform the operations reported above.

Chapter 2

Background

We will begin this chapter by looking in more detail at how Orange was created, how it evolved into a powerful visual programming application for data mining, and its current features and applications. As mentioned in the introduction, the workflows used to perform data mining within the Orange application are actually directed acyclic graphs. The properties of these mathematical tools are essential for understanding what information is sufficient and necessary to describe them unambiguously. Therefore, a background on graphs is given with special emphasis on the topological sorting operation, which will allow us to create a sorted representation of how the widgets interact with each other. This piece of information is essential in order to make the LLMs ”understand” the data mining processes implied by the Orange workflows.

To make this project possible, a dataset of possible data mining workflows has been created by analyzing the screenshots taken from the lectures intended to teach students how to use Orange. A series of image processing techniques were used to extract the workflows from these screenshots: the theoretical background behind these operations is also provided in this chapter.

The created dataset is then further analyzed to gain insight into the similarity of workflows and the importance that widgets have inside them. The former is obtained by applying the t-distributed Stochastic Neighbor Embedding (t-SNE) to a vectorial representation of the dataset workflows, while the latter is achieved through the application of the hypergeometric distribution, which is necessary to understand which widgets are strictly related to a given lecture chapter. The theoretical background behind these operations is therefore provided in this chapter.

Finally, since this study is entirely based on the use of Large Language Models, information is given on how these models are built and set up to work, as well as brief explanations of

the prompt engineering techniques used in this thesis. This piece of information is essential and represents the basis behind many of the choices taken in this study, especially since the understanding of workflows and graphs is a weak point of the LLMs, as proven by the studies cited in the introduction.

2.1 Orange Data Mining Toolbox

Orange Data Mining is a powerful tool that allows data scientists to analyze different types of data sets. Originally built as a Python toolbox for machine learning and data mining, it relies heavily on the implementation of C++ computationally intensive tasks and is one of the oldest tools in its field, having been created in 1997. Similar to *scikit-learn* and *mlpy*, this toolbox has the distinctive feature of combining symbolic, string, and numeric attributes and metadata information, thus facilitating machine learning analysis on multimodal datasets. The main strength of Orange is that it is hierarchically organized, which means that different levels of procedures are implemented. Lower-level procedures can be used to create higher-level ones, allowing users to construct any functionality they desire by assembling different components.[4]

While Orange is an extremely useful Python toolbox, it used to be inaccessible to non-programmers. This was a problem because the group that created Orange has a tradition of collaborating with partners from other scientific and industrial fields, with a particular focus on biomedical applications. Therefore, there was a need to implement a visual programming method to use Orange's powerful components, and thus the Orange Data Mining application was born. The Python library *Qt* was chosen to create a graphical user interface, which is now the primary way most users use Orange. Similarly to the way the toolbox is structured, the visual programming also uses different work units, called widgets, and the connections between them to achieve the desired operations, allowing users to construct workflows based on their specific needs. [5] Currently, there are 227 widgets available, organized into different modules such as Data, Transform, Model, Text Mining, and Image Analytics. From now on, widgets will also be reported using the Module/Widget Name format.

2.1.1 Workflows

As mentioned in the previous section, workflows are the main strength of Orange visual programming. Widgets are capable of performing powerful operations and have a range of possible inputs and outputs. Combining widgets in different ways can lead to different types of analysis that users can customize to their needs. In a way, this is both a strength and a weakness. Because there are so many ways to connect the widgets, and because changing the connections

between them can significantly change the type of operations performed on the dataset, this type of programming is not the most beginner-friendly solution. Orange is therefore a tool with great potential, but also with a really steep learning curve.

Widgets are not only operational units; most of them also allow for either model or data visualization. They all have different settings that can be modified according to the user's needs. Many widgets also allow you to select subsets of data instances for further analysis, further complicating the technical nuances of this application. For clarity, the type of output-input pair is reported above the link in question, giving the user immediate feedback on how the data is being passed along in the workflow. As shown in figure 2.1, in cases where there is an inconsistency between the output-input pairs, the program automatically detects it and reports it with a dashed line. Warnings and errors are also reported in the workflows (see the error reported as a red exclamation mark above the Model/Logistic Regression widget and the warning reported as a yellow exclamation mark above the Text Mining/Import Documents widget in Figure 2.1), giving users an immediate visual representation when something is not working in the workflow. By hovering over errors and warnings, the program provides more information about what went wrong, making it easier for the user to identify and correct the mistake.

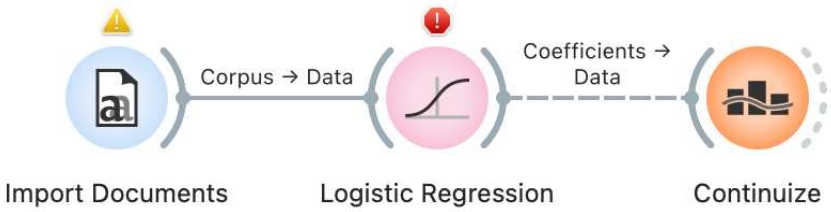


Figure 2.1: Example of mistakes in a workflow

The large number of widgets available in the Orange visual programming application allows us to perform many types of operations and analyses, but the novice user will probably use this program mainly to perform classification and exploratory data analysis on the built-in datasets. These types of analyses are quite powerful and efficient, but they are just the tip of the iceberg of what Orange can do.

There are many types of datasets that can be loaded from the repositories or from a file. From the available widgets and repositories, the datasets that can be loaded are either tables, images, documents, single cell data, or gene datasets. These datasets can then be preprocessed in a variety of ways to highlight certain features or emphasize certain types of analysis. Tabular data sets can be used to perform exploratory, classification, and any type of distance-based analysis (e.g., clustering and principal component analysis), as well as signal and spectrum-based analysis on signals. The tabular datasets can also be transformed into survival data, for

which a small set of analyses is implemented in Orange. For image and document datasets, most of the above can also be done thanks to the Image Embedding and Document Embedding widgets, which allow us to transform the respective datasets into vectorial datasets that can be used for the above-mentioned analyses. For document datasets, a number of text mining techniques are also implemented in Orange so that documents and corpora can be analyzed in more depth. Single cell datasets also have a small set of basic operations that, when implemented with the gene analysis ones, allow most of the essential operations to be performed to understand and extract data from single cell datasets. This shows that these data types are not completely separate, but can be interchanged and used in the same workflow, depending on what the user is trying to accomplish. We will now provide some examples of commonly used workflows with an explanation of their functionality.

The workflow shown in Figure 2.2 focuses on comparing the performance of three different classification models: Support Vector Machine (SVM), Random Forest, and Logistic Regression. It loads a data set from a file, which is visualized in a data table. The data set is then used to train and score the three models. The Test and Score widget evaluates the performance of each model and provides results that are further visualized in a Confusion Matrix. This matrix allows for detailed analysis of the models' predictions, highlighting instances of correct and incorrect classifications.

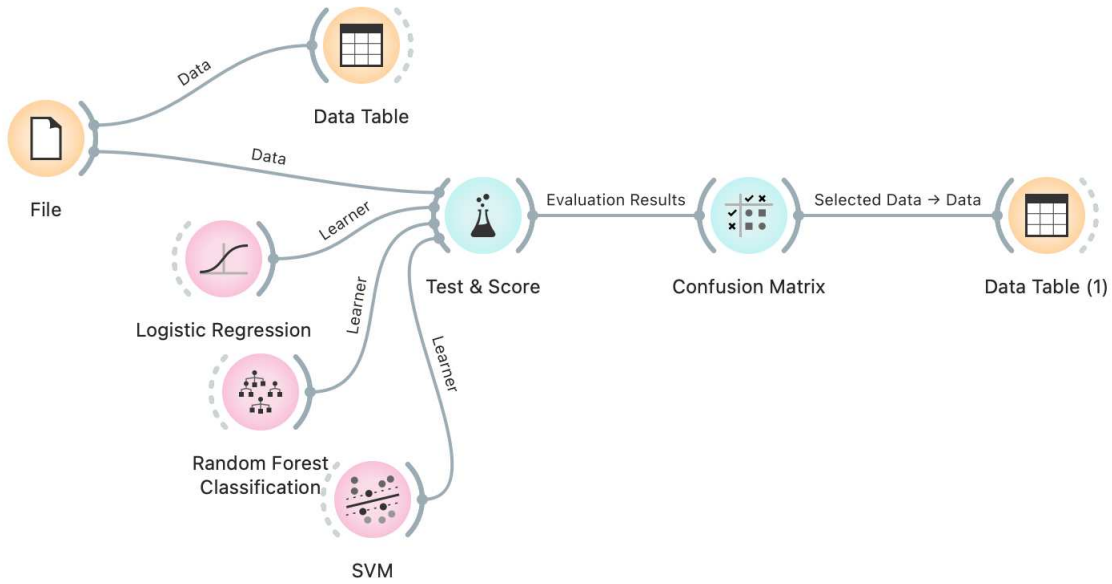


Figure 2.2: Workflow that implements classification on tabular data

The workflow represented in Figure 2.3 begins by importing images from a directory. It then extracts image embeddings (feature vectors) using deep learning models. These embeddings are used to calculate distances between images, which are then input into a hierarchical clustering

algorithm. The selected clusters, along with the original image data, are presented in a data table for further examination and analysis.

Figure 2.3: Workflow that performs clustering analysis on images

The workflow represented in Figure 2.4 processes a Corpus of documents. It begins by preprocessing the corpus, transforming it into a Word Cloud to display word frequencies. The user can select words from the word cloud, which are then used as queries in the Concordance widget to find their occurrences in context. The selected documents from the Concordance widget are then visualized in the Corpus Viewer for further analysis.

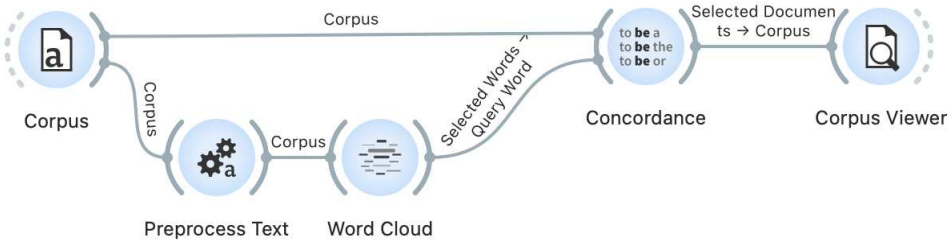


Figure 2.4: Workflow that performs text analysis on a corpus

The workflow represented in Figure 2.5 analyzes single-cell datasets. It first loads the dataset and reduces its dimensionality using t-SNE for visualization. Then, it annotates the cells using Marker Genes data, assigning cell types based on gene expression. Finally, the annotated data is displayed in a Data Table for further analysis.

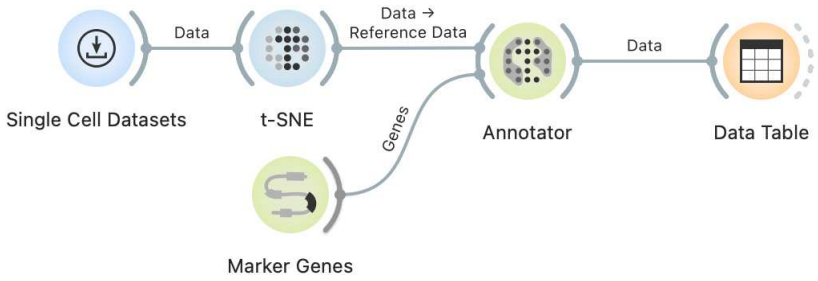


Figure 2.5: Workflow that performs gene expression analysis on single cell data

2.1.2 Graphs

Since an Orange workflow can also be expressed as a graph, specifically a directed acyclic graph, where each widget is a vertex and each link is an arc, we will now describe the main properties of such objects.

A **graph** is a network of vertices and edges. The vertices represent the objects present in the network and the edges define how these objects are connected to each other. A directed graph, or **digraph**, is a graph in which the edges have orientations (see figure 2.6). **Directed Acyclic Graphs (DAGs)** are special digraphs that do not allow directed cycles, meaning that from any vertex, by following the arcs and their direction, you never go back to the vertex you started from. This type of graph is the perfect representation for Orange workflows because it represents exactly how they are built. Each widget has inputs and outputs, and it is impossible to create cycles, so a connection between two widgets can be interpreted as an ordered pair of vertices. [6]

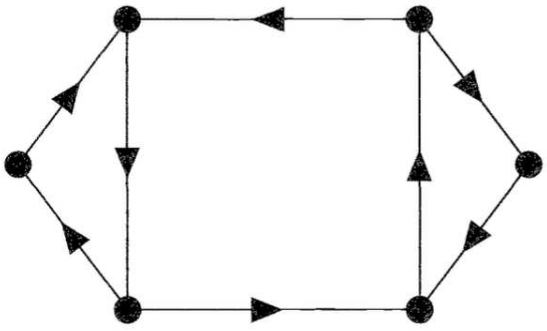


Figure 2.6: Example of directed graph

The definition given for DAGs is really important because it allows to find the order in which the nodes are positioned inside the digraphs, thanks to **topological sorting**. This is essential in our case because, as reported in the introduction, the ability of LLMs to understand graph information is not good and strongly depends on the approach used to provide the information to the model. While conducting experiments on the best prompts to use for each task, we found that the order in which the links are reported strongly affects LLM's understanding of the workflow and the results. Given a DAG, the goal of topological sorting is to find a linear order of vertices such that every tail of every arc comes before its head in the order. This can be done with many different algorithms, in our case we will use the one from the *graphlib* Python library. [7][8]

2.2 Image Processing

As mentioned above, Orange workflows are acyclic digraphs. Each node is a widget, and almost (see figure 2.7) each widget has a unique recognizable icon. In addition, the widgets are connected by a line built with splines that goes from the right antenna near the widget (output antenna) to the left antenna of another widget (input antenna). To detect the widgets from the screenshot, the following image processing operations are performed: Circle Hough transform, edge detection, and calculation of the normalized correlation coefficient. The link detection process uses the following techniques: thresholding, connected component detection, and bit-wise AND operation. We will now provide the theoretical background for these techniques.

(a) Model/Tree widget icon (b) Text Mining/Ontology widget icon

Figure 2.7: Example of identical widget icons

Circles The Hough transform (CHT) is used to identify the circles in the image. In general, the Hough transform is used to define a mapping between an image space and a parameter space. In the case of the CHT, the parameter space is 3-dimensional and consists of the two coordinates of the center of the circle and the radius of the detected circle. Basically, for each pixel in the image space, a circular cone is created in the parameter space, resulting in a voting process for the possible parameter trios. This type of operation is usually performed on binary images, which means that an edge detection operation is usually performed before the CHT is applied. [9] **Edge Detection** consists of a process that detects boundaries between regions of different intensities, in the case of grayscale images. This can be done in many ways, for example, by calculating the gradient of the image and using it to define the slope and direction of the surface, or by using the Canny edge detector, which is a hysteresis-based thresholding operation. [10]

The **Normalized Correlation Coefficient** metric is computed and used to perform template matching on images. Each template is used as a moving image and checked against the reference image. The Normalized Correlation Coefficient is calculated using the following formula:

$$R(x,y) = \frac{\sum_{x',y'}(T'(x',y') * I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 * \sum_{x',y'} I'(x+x',y+y')^2}}$$

where T' is the template (i.e., the widget drawing), I' is the reference image (i.e., the screenshot of the workflow), x and y are the pixel locations in the reference image for which the correlation coefficient is calculated, and x' and y' are the pixel locations from the moving image (i.e., considering a widget drawing of size 43-by-43 pixels, $x' = 0, 1, 2, ...42$ and $y' = 0, 1, 2, ...42$). In this formula, both the moving and the reference images are transformed into zero-mean images, which means that bright pixels are positive and dark pixels are negative, and a dissonance in intensity level between two matching pixels decreases the correlation coefficient, while a concordance increases it. Thanks to these operations, for each widget icon, we get an array of size $screenshot\_width - icon\_width + 1$-by-$screenshot\_height - icon\_height + 1$, where each value in the output array is the normalized correlation coefficient calculated from the relative pixel in the screenshot. [11][12]

To identify the links, thresholding and connected component identification are performed. **Thresholding** is the process of converting a grayscale image into a binary image by implementing a chosen threshold. Every pixel with an intensity lower than the chosen threshold is converted into a black pixel (i.e., intensity equal to 0), and every pixel with an intensity higher than the threshold is converted into a white pixel (i.e., intensity equal to 255 or 1, depending on how the grayscale is set up). Since we need to find the connected components of the links, the pixels belonging to the latter must be set to white in the binary image, i.e. we use inverse binary thresholding, thus turning dark pixels (with intensity lower than the threshold) into white pixels and bright pixels into black pixels. This allows us to perform analysis on the **connected components** in the binary image. A white pixel is 4-connected to another white pixel if the latter is adjacent to the pixel in question, as opposed to an 8-connection between pixels, which also includes the possibility that two widgets are diagonally adjacent to each other. Once the clusters of connected pixels are identified, a label is assigned to each cluster, thus identifying all the connected components present in the binary image. Finally, the **pixel-wise 'and'** operation is used to extract the workflows from the screenshots. Given two binary images of the same size, pixel-wise *and* of the two images will output an image with the same dimensions as the original images. In this image, a pixel will be white if the same pixel was white in both original images, otherwise it will be black. Therefore, this operation will only highlight the areas in the resulting image that are white in both original images and turn everything else to black. [10]

2.3 Workflows similarity analysis

As mentioned earlier in the chapter, the reference workflows dataset used in this study is built through the analysis of lecture notes. In these notes there are many different workflows that achieve different things, thanks to their unique widgets and structures. To assess the similarity

of different workflows coming from the lectures, t-SNE visual representation is used. This allows us to have an immediate visual representation of how workflows coming from the same lecture chapter could be highly related to each other and form a cluster or, on the contrary, be very close to the ones of other chapters and, therefore, create an highly intertwined system. Furthermore, to gain insight into the composition of workflows and the importance of widgets inside them, hypergeometric distribution is used to identify enriched components within a given lecture chapter.

t-distributed Stochastic Neighbor Embedding (t-SNE) is a machine learning algorithm typically used to perform dimensionality reduction of high-dimensional datasets. By converting the Euclidean distances between data points into conditional probabilities this algorithm allows to project the data points on a two-dimensional plane, thus allowing a simple visualization of the similarity between different data points as well as the emergence of possible clusters of data. The conditional probability $p_{j|i}$ is calculated using the following formula:

$$p_{j|i} = \frac{\exp\left(-||x_i - x_j||^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-||x_i - x_k||^2 / 2\sigma_i^2\right)}$$

where x_i is the i-th data point and σ_i is the variance of the Gaussian that is centered on the i-th data point. Once the probabilities are calculated, the data points are shown on the two-dimensional plane in such a way that two data points with low conditional probability are shown far apart and two data point with high conditional probability are shown close together.[13]

The **hypergeometric distribution** is a statistical tool that, given a fixed number N of objects present in a finite set and a fixed number of objects k for each different kind belonging to that finite set, allows us to calculate the probability of finding x objects for a chosen kind in a sample of size n. This probability is calculated using the following formula:

$$P(X = x) = \frac{\binom{k}{x}\binom{N-k}{n-x}}{\binom{N}{n}} \text{ where } \binom{n}{k} = \frac{n!}{k!(n-k)!} \text{ [14]}$$

This distribution is important to recognize which feature is enriched in a dataset, meaning that it allows us to understand which features have a significant importance for the description of the dataset.

2.4 Large Language Models

In recent years, the world of conversational Artificial Intelligence (AI) has changed dramatically with the rise of neural networks. Neural networks built with the sole purpose of processing human language are called language models. As mentioned before, these models are the

foundations on which this entire study is based on, and, as such, their theoretical background, provided below, is of great importance.

Recently, these types of models have started to be trained with more and more parameters and on larger and larger data sets. A few years ago, it was believed that increasing the size of a neural network too much would degrade its performance, but this was disproved by the efficiency of **Large Language Models (LLMs)**. In fact, these models proved that using more data and more computing power to train models with more parameters resulted in better performance. An LLM is a language model that has a really large number of parameters. Experts still disagree on how many parameters a model needs to be considered large, but the best performing models today have thousands or even millions of parameters, compared to models trained ten years ago. [15]

LLMs are models specifically designed to understand and generate human language, but their ability to understand and generate is not only limited to text and language; in fact, there are several LLMs that are also able to understand and generate images (e.g. DALL-E). To accomplish such tasks, these models use a combination of neural networks (in most cases, transformational neural networks) and machine learning algorithms to process text and images in a manner similar to that of a human. Thus, the goal of a researcher building an LLM is to model a digital representation of language that might be useful for performing conversational tasks on the input dataset. These models can understand the dependency of words and sentences, and the output is produced by writing the word that has the highest probability of coming next in a sentence, given the text that preceded it. The more powerful a model is, the better it can recognize dependencies and relationships between different parts of the text. The main strength of LLMs is that they can be used to perform many natural language processing tasks because they understand the structure of text more broadly and are not trained on a specific task, making them very versatile. [16]

2.4.1 Chatbots

LLMs, as their name implies, are simply models, and for them to be used, there needs to be a interaction point between them and the end user; this role is fulfilled by chatbots. A chatbot is defined as a computer program that simulates conversations with an end user. This type of technology is often used within applications or websites to help end users to use and understand the services and features provided, thus improving the user experience. This function is exactly what we aim to achieve with the creation of the virtual assistant presented in this study. We will now look at the theoretical underpinnings of most existing chatbots: natural language processing.

Natural Language Processing (NLP) is the ability of computers to understand statements or words written in human language. It is divided into two parts: Natural Language Understanding (NLU) and Natural Language Generation (NLG). The former aims at making the machine understand what is said in a sentence or document, enabling it to extract concepts, entities, emotions, keywords, etc., while the latter focuses more on the process of generating text that has meaning from an internal representation. [17]

Natural Language Understanding (NLU), also known as linguistics, is the study of the meaning, context, and forms of language. It is divided into seven different levels of language analysis. The first is **Phonology**, which refers to the systematic arrangement of sounds, thus focusing on the sound part of languages, especially on their behavior and organization. **Morphology** studies the structure of a word and its smallest units of meaning, known as morphemes. The combination of morphemes used to form a word gives it its meaning. **Lexical** is used by both humans and NLP systems to interpret the meaning of individual words, taking into account the context in which they occur. It pays special attention to the Part of Speech (PoS) tagging of words. In order to categorize different words, some operations are performed to remove meaningless parts of the text. The main operations are stop word removal (removing words like 'in', 'the', 'and', etc.), stemming (extracting the root of a word, e.g. turning 'dramatic' into 'drama', thus removing the suffix), and lemmatization (similar to stemming, except that it does not remove the suffix, but rather returns the basic form from the vocabulary; for example, the lemmatization of 'using' would be 'use', whereas the stemming would be 'us'). The **syntactic** level takes the output of the PoS tagging and analyzes the grammatical structure of the sentence, primarily word order and dependency between words. Since the meaning of a sentence can change depending on prepositions and word suffixes, the text processing techniques used for the lexical level are not applied. **The semantic level** determines the meaning of a sentence. To be efficient at this level, machines determine the most relevant word in a sentence and analyze how other words interact with it. **Discourse** analyzes more than one sentence and is concerned with the analysis of logical structure, making connections between separate sentences that ensure the coherence of the text. Finally, there is the **pragmatic** level, which focuses on understanding something that is not openly stated in the sentence or text. A given sentence may have different meanings when used in different contexts, and pragmatic analysis focuses on identifying which of the possible meanings is the correct one. [17]

As mentioned above, **Natural Language Generation (NLG)** focuses more on the process of generating text. This is achieved by going through four different phases: goal identification, situation evaluation, evaluation of available communication sources, and text generation. NLG consists of three components. The first is the **application or speaker**, which is used to maintain

the situation model, the second is the **components and levels of representation**, which is further composed of four different components: textual organization, selection of content and linguistic resources, and realization. The last component of NLG is the **speaker and generator**, which is needed to transform the intentions of the application into text relevant to the situation. It stores important information that may be relevant for future use in text generation. [17]

2.4.2 Prompt engineering

The advent of LLMs is a great opportunity to perform human language tasks faster and better. Their ability to process text far exceeds that of normal humans, but in order to be used efficiently, the requests made to the model must be made in a way that is understood by the LLM. Thus, prompt engineering has recently begun to be studied. The ability of these models to understand context and word dependencies is so good that new techniques have been created and studied to make them perform different convoluted tasks. We will now look at the techniques that are most useful in a one-prompt scenario, i.e. not in a conversational style of use. [18]

In the **persona pattern**, the input given to the LLM begins with a statement in which the user instructs the model to act as a particular type of persona. The selected persona can be of any type; for example, it could be a job description, a fictional character, or a historical figure. This is particularly useful for making the LLM automatically adapt its output to the specific needs of the task the user is trying to accomplish. An example of this pattern might be *"You will pretend to be Christopher Columbus..."* or *"Answer like Christopher Columbus would..."*. It is important to note that in cases where the chosen persona could be considered harmful, is a living person, or is under copyright, the LLM will probably ignore the request. It is also important to note that in certain scenarios the chosen persona may cause the LLM to make its own assumptions about the context, and this must be taken into account when creating the prompt. [19]

In the **metalanguage creation pattern**, the prompt contains a newly created pattern in which information about the context is given to the LLM. This allows the user to create an alternate language or scheme to provide information to the LLM. This is particularly useful for creating a shorthand notation to describe graphs or other unstructured scenarios that can be described by a recurring pattern. An example use of this pattern might be

"I am now going to give you information about a digraph. Whenever I write Node A -> Node B, it means that there is a link from Node A to Node B in the digraph." [19]

In the **template pattern** the created prompt will contain information about the desired format of the output coming from the LLM. To achieve this result, the user can specify a format that

contains placeholders for the LLM to replace when generating the response. These placeholders can be specified in many ways, the most common being the use of uppercase letters and the use of brackets like this: <placeholder>. An example of use might be as follows:

"The following is the format that your output should follow; replace the PLACEHOLDERS with your output:
The chosen item is CHOSEN ITEM with a probability of PERCENTAGE PROBABILITY of being the right one." [19]

Zero-shot prompting consists of directly telling the LLM what tasks we want it to perform. It is therefore one of the simplest prompts a user can create, but it can still produce pretty good results. The more detailed the request, the better, and the closer the output will be to what the user wants. With this technique, the LLM, if additional information to the prompt is used to perform the task, then the model will use that as well to provide an output, otherwise only the text used for training will be considered to generate the output. An example of using this technique would be

"Write an article explaining what black holes are." [18]

The **few-shot prompting** technique is to give the LLM a few high-quality input-output examples to help it understand what we want it to do. This is both a powerful and a dangerous way to create a prompt, because if the examples are well done, it can greatly improve the quality of the output, but these examples can also introduce an unwanted bias into the solution of the task. Furthermore, in a world where each token must be paid for, increasing the number required for each prompt also increases the cost of each prompt-output pair. An example use of this technique might be:

"Q: Car
A: Fast

Q: Snail
A: Slow

Q: Cheetah
A: Fast

Q: Rock

A: Slow

Q: Bullet
A:" [18]

Chain-of-Thought (CoT) prompting consists in providing a rational outline for the solution of a given problem. The user is thus shown how a problem can be made easier by a logical chain of reasoning, and the LLM will learn and imitate this type of process to solve future problems. This processing sketch will help the model to decompose a multi-step problem into intermediate steps and provide feedback to the user on how the model arrived at the conclusion given by the output. An example use of this technique might be

"Q: Given the numbers 3, 12, 31, 15, the sum of the odd numbers in this sequence is even
A: The odd numbers are 3, 31, and 15, and their sum is 49. FALSE

Q: Given the numbers 5, 21, 17, 29, the sum of the odd numbers in this sequence is even
A:"

Since CoT provides examples of rational contours, it can be seen as a special version of few-shot prompting. However, this technique can be implemented without providing a logical reasoning example, and similar results can be achieved by simply adding *"Let's think step by step."* at the end of the prompt. This way of using CoT is called **zero-shot CoT**.

Chapter 3

Methods

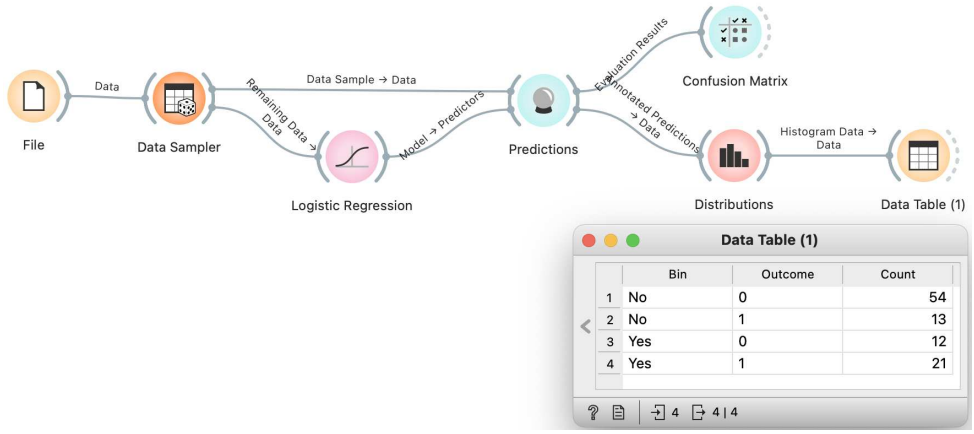
As mentioned before, many of the experiments performed in this thesis use as an information base the dataset of workflows obtained from the lectures containing screenshots of the Orange application. In this chapter, we will see what operations were performed to extract the workflows from the screenshots, the methods used to visualize the resulting dataset, and finally the prompts for the LLMs to perform the tasks.

3.1 Screenshot Analysis

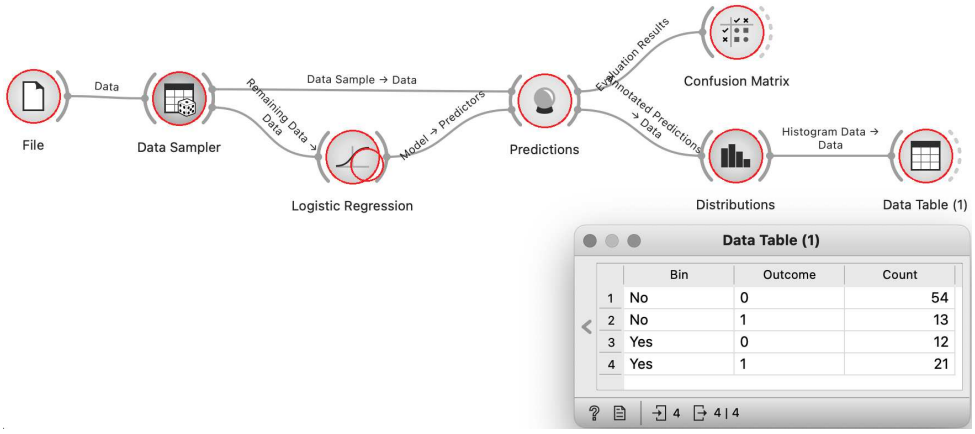
In order for the LLMs to perform the tasks we set out to accomplish, we needed a way to infer information about the Orange workflows and how they work. This is accomplished by using workflow screenshots, which are present in a dataset consisting of 757 screenshots, of which only 308 actually contain workflows. This dataset comes from a series of notebooks created to teach university students about Orange's functionalities. In order to analyze the screenshot dataset and extract information about the workflows, a series of image processing operations are performed on each screenshot. Since they were created by different people over the years, they all have different characteristics: they vary in the resolution of the widgets, with widths ranging from 30 pixels to 120 pixels, they have different layouts, and some of them are old enough to contain obsolete widget icons. We will now see the operations performed to extract the workflows from the screenshots.

The first step needed to extract the workflows from the screenshots is to identify which widgets are present in the workflow and where they are located. This requires parsing each screenshot and downloading the widget icons from the Orange widget catalog [20]. In image processing, there are many techniques to find an object in an image; most of them are useful for

scaling variations, i.e. a way to detect the size of the widgets in a given screenshot is needed to match the size of the downloaded widget icons (which are 100 pixels wide) with their counterpart in the workflow. Although the screenshots are all colored, they are converted to grayscale images when they are loaded to make it easier to perform operations on them.



(a) Original screenshot



(b) Circles detected through CHT are drawn in red

Figure 3.1: Example of circle detection used on notebook workflow

Since the widget icons are circles, the diameter of the circles found in a given screenshot is expected to be the same as the size of the widgets in the image. This is not exactly true, because as we can see from 3.1.b, the circle detection process is not perfect. The function used to detect the circles is taken from the OpenCV Python library and allows the user to tune many parameters to get the desired results. This function first applies the Canny edge detector, then detects the circles using CHT, and finally applies a threshold to select circle candidates with an accumulator value above it. Since the function performs edge detection, it is recommended to blur the image before passing it to this function, otherwise high frequency noise will be amplified, potentially leading to false circle detection. The main tuning parameters are

- *minDist*: minimum distance between detected circle centers
- *param1*: threshold used by the Canny edge detector
- *param2*: accumulator threshold used for circle candidate selection
- *minRadius*: minimum circle radius to detect
- *maxRadius*: maximum circle radius to detect [21]

Since our screenshot dataset is very large, it is almost impossible to find a set of parameters that works for every image in it. In our use case, we set a minimum radius size of 15 pixels and a maximum radius size of 60 pixels to reduce the risk of detecting circular objects that are not widgets (this is important because there are screenshots like the one shown in 3.2 that do not contain workflows but do contain circular objects). The minimum distance between circles is set to 30, because there are screenshots where the widgets can be as small as 32 pixels, and so the theoretical minimum distance between two widgets is 32 pixels. *param1* and *param2* were arbitrarily adjusted by trial and error and ended up being 65 and 55 respectively. The pseudocode for the algorithm that detects the size of the widgets in the screenshots is reported as algorithm 1. [21]

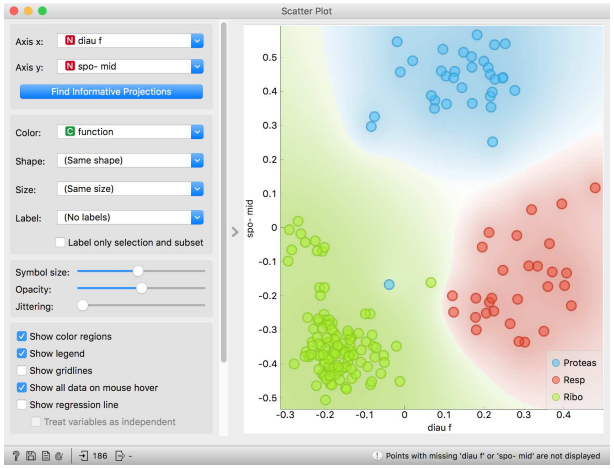
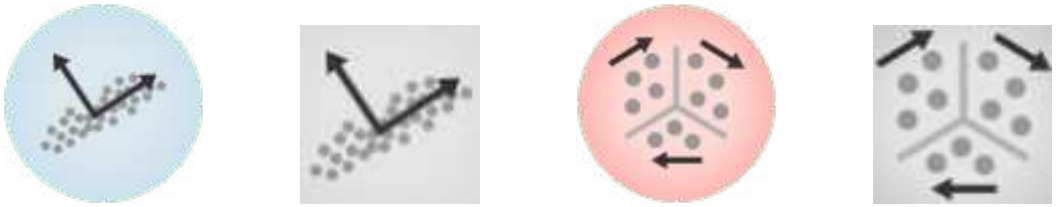


Figure 3.2: Example of a screenshot that does not contain a workflow but contains circular shaped objects in it

Algorithm 1 Get widget size from screenshot

Load the screenshot as a grey-scale image
Apply Gaussian blur to the screenshot
Identify circles in the screenshot through Hough Circle Transform and extract their $radii$
widget size $\leftarrow round(median(radii * 2) + 1)$

Once the screenshot widget size is identified, template matching is performed. To do this, all widget icons are loaded from the widget catalog, cropped to include only the drawing in the icon, and scaled so that the new size matches the size identified in the screenshot (see Figure 3.3). Once the widget icons are loaded, template matching is performed using the Normalized Correlation Coefficient.



(a) Original icon (b) Cropped drawing (c) Original icon (d) Cropped drawing

Figure 3.3: Example of widget drawings used for template matching and obtained by cropping the original icons

A threshold is then set to identify a potential match. Given that a perfect match would give a normalized correlation coefficient of 1, and that the matches should theoretically be close to perfect since the workflows use the icons taken from the catalog to identify the widgets, one might think that a threshold very close to 1 would work just fine. Unfortunately, this is not the case because the latter assumption is not always true. We are not in an ideal scenario where all screenshots have the same resolution, and since the widget icons are scaled, the 1-to-1 pixel correspondence is not perfect, so the matches are almost never perfect. Although interpolation helps to create a really close match between the cropped icon and the drawings in the screenshot, if we choose a threshold that is too high, there is a really high probability that many widgets would not be identified in the screenshots, which would cause problems in workflow identification. Considering this fact, a threshold of 0.8 is chosen. This guarantees that basically all widgets would find a successful match if they are actually present in the screenshot. However, this leads to the opposite problem, i.e. if two icons are similar to each other, a false match could be identified. This is fixed by considering the location of the widgets identified by the template matching: once the locations of all matches are found, if there are overlapping matches, only the widget related to the highest normalized correlation coefficient are kept, thus eliminating potential mismatches. Additionally, the distance between same-widget matches is implemented so that for each widget, only one match would be considered if the distance between two identified locations is less than the widget size.

To allow link detection and further operations, information about the matches and their locations is stored in an array of size $n\_widgets$-by-5. Each row contains information about the corresponding widget. The first column contains information about the presence of the widget.

This is especially important if a workflow has multiple instances of the same widget. Since there is only one row for each widget, a way is needed to write information about possible repetitions in a different way. If a widget is present, this column would represent a 1 in the respective row, but if the row is already occupied, this field would represent a negative number in one of the previous rows. The degree of the number indicates which line it refers to (e.g. if the widget corresponding to line n. 120 is present twice, the first time it would be stored in line 120 with a 1 in the first column, and the second instance would be stored with a -1 in line 119 if it is free, or with a -2 in 118 if line 119 is already occupied, or so on until a viable line is found). The second, third, and fourth columns contain information about the location of the found match, with the second column reporting the index of the found match in the ”flattened” array, and the third and fourth columns reporting the width and height, respectively, of the array obtained by template matching. Finally, the fifth column contains the normalized correlation coefficient for the found relative match. The pseudocode for this algorithm is given as algorithm 2. As can be seen from the pseudocode, there are some special widgets that are treated differently, i.e., they use custom thresholds to limit potential errors in the widget detection process. A lower threshold is used for widgets that are sensitive to small changes in icon representation, while a higher threshold is chosen for widget icons that have caused false positives. In addition, a problem with the Model/Tree widget detection occurred because the same icon is used for the Text Mining/Ontology widget, so since the latter never appears in the dataset, the ability to detect this widget was completely removed.

Once the widgets present in a screenshot and their locations are found, the next step needed to characterize the workflow is link detection. This is tricky because there is no rule that controls how widgets are organized in a workflow, and because links are drawn using splines, making the line Hough transform useless in this scenario. Since the links in the screenshots are always drawn with the same color and are continuous lines, an easy way to identify the links would be to threshold the image and use an algorithm that finds connected components in a binary image. This works effectively and guarantees that the links are correctly identified and labeled, provided that the connection lines are not interrupted in the screenshot. Since we know that, given a widget, all incoming links are connected to the antenna located to the left of the widget and all outgoing links are connected to the antenna located to the right of the widget, and since we know where the widgets are located in the screenshot, we can identify which connected components are representative to the input and output of each widget.

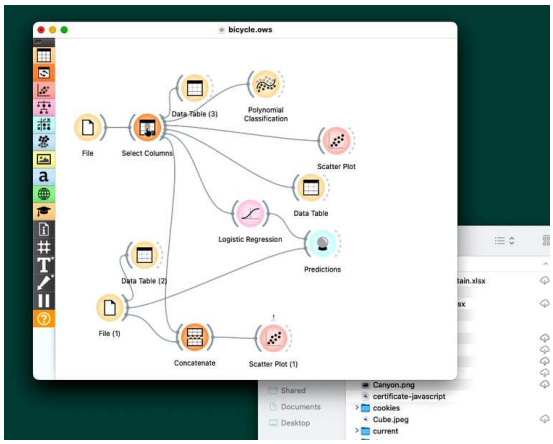
However, this is not enough to identify the links between widgets, because there are many workflows that have either overlapping links or closed loops. This causes the connected component of a link to merge with those of other links, meaning that the same connected component

Algorithm 2 Widget location array creation

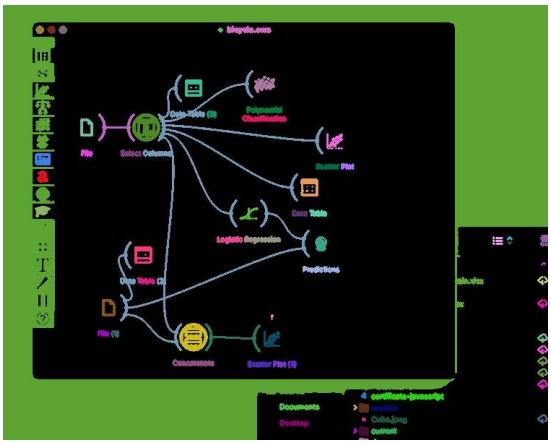
Set a value for the threshold th
Load the screenshot and all the widget icons
Initialize as zeros an array of size $(N_{widgets}, 5)$ where to save the detection
for Each widget icon **do**
  Calculate the normalized Correlation Coefficients ($ccoeff$) for the widget on the screenshot

  if The widget is Data/Datasets, Data/File, Educational/Polynomial Regression, Transform/Impute or Unsupervised/Distance File **then**
    $th \leftarrow 0.7$
  else if The widget is either Time Series/Correlogram, Time Series/Periodogram or Bioinformatics/Dicty Express **then**
    $th \leftarrow 0.9$
  else if The widget is Text Mining/Ontology **then**
    $th \leftarrow 1$
  else
    The threshold stays the same
  end if
  Find how many coefficients are above th
  if There is at least one coefficient greater than th **then**
    Order the coefficients so that we can save them in descendent order
    Initialize a location vector for each axis to save where the current widget has been found

    for Each coefficient greater than th **do**
      if The location of the found widget is far from previous ones **then**
        Save the widget's presence (1st dimension, it is 1 if the widget is saved on its line, $-$*line number difference* otherwise), location (dimension 2), output shape (dimensions 3 and 4) and the found *ccoeff* (dimension 5) in the array
        Append the found widget axis locations to the axis location vectors
      end if
    end for
  end if
end for
$j \leftarrow 0$
Calculate $N_{detected\ widgets}$
while $j < N_{detected\ widgets}$ **do**
  Find locations of the detected widgets
  if Some widgets overlap **then**
    Find which one of the overlapping widgets has the highest *ccoeff*
    Delete the match for the widgets with lower *ccoeff*
    Calculate $N_{detected\ widgets}$
  end if
  if The widget deleted wasn't the j-th **then**
    $j \leftarrow j + 1$
  end if
end while



(a) Original workflow



(b) Different connected components highlighted with different colors

Figure 3.4: Demonstration of labeled connected components

that connects two linked widgets may also connect widgets that should not be linked to them (see figure 3.5). To solve this problem, a special algorithm has been created to handle these cases.

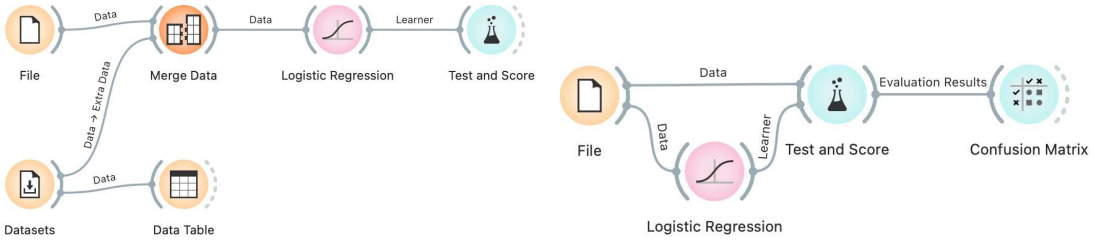


Figure 3.5: Example of workflows in which different widgets share the same connected component

The main link detection algorithm is based on a smaller algorithm that detects intersections between the original binary image and an artificially created circle. Once the center and radius of the circles are provided, the resulting white circle is drawn on an image of the same size as the original. The two images are then used to perform a pixel-by-pixel *and* operation to find the intersections between the constructed circle and the supposed link. A connected components analysis is performed on the resulting image to identify the different clusters of pixels found. Furthermore, this algorithm is also used to detect the direction of the found clusters relative to the center of the circle, which will prove essential later when the link detection algorithm is presented. The resulting pseudocode for this algorithm is given as algorithm 3.

Once the *find_circle_intersection* function provided above is defined, we can begin to describe the link detection algorithm. After labeling the connected components from the thresh-

Algorithm 3 Find circle intersection

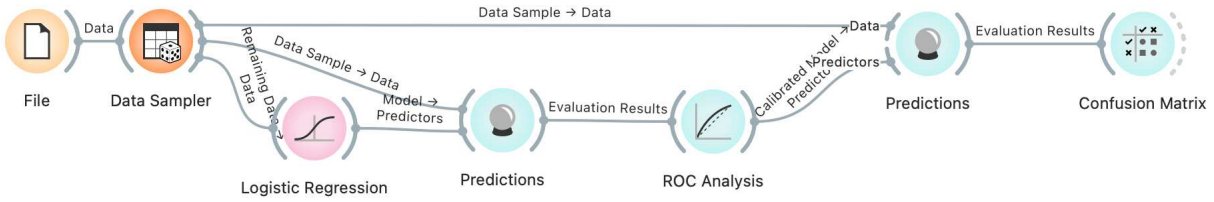
Set the values for the center location, the radius of the circle and the previous direction
Create a binary image with the same size of the screenshot that contains a white circle of the given center and radius
Perform binary pixel-wise and between the circle image and the thresholded screenshot image

Find connected components on the obtained image and find how many cluster of points were found
for Every found cluster of points **do**
  Calculate the average position of the cluster pixels to find its center
end for
Given the points found at the previous steps calculate their direction relative to the center of the circle with the inverse tangent function

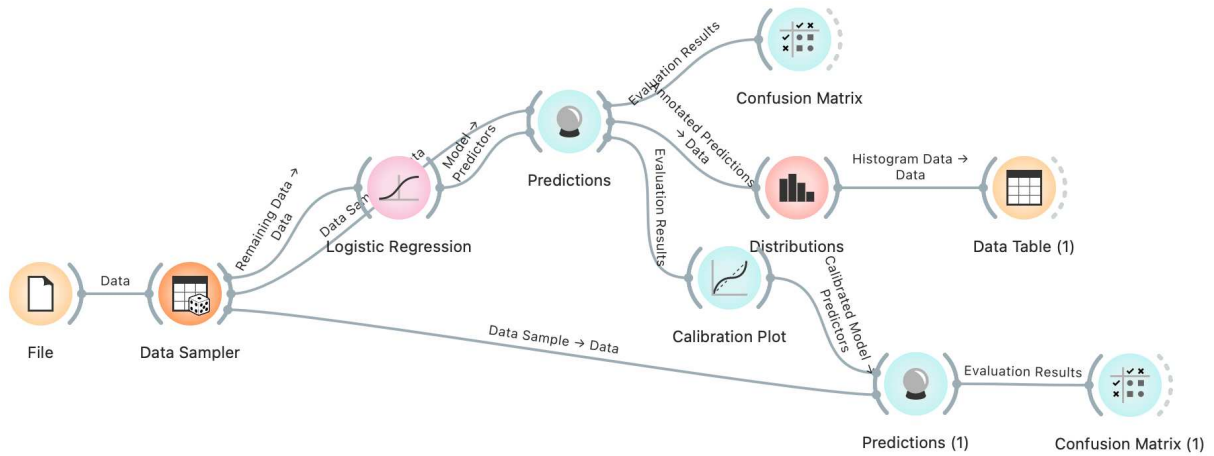
olded screenshot and collecting the widget location array, we know which labels are near the widgets. We know that an output link will be to the right of a widget and an input link will be to the left of a widget, so we start looking for which labeled component appears both slightly to the right of one widget and slightly to the left of another. Once we have found the connected components that satisfy this rule, we begin to examine them. If said component creates a single pair of connected widgets, then it represents the link between the two widgets and no further operations are necessary. If this is not the case, then the component needs to be checked. We start by identifying which widgets use it for input links. For each of the found widgets, the circle intersection algorithm is used by creating a circle with center slightly to the left of the widget and radius equal to that of the widget, while the reference binary image used is white in the positions where the connected component of interest is found in the original image. In ideal conditions the number of intersections found will be equal to the number of incoming links. For each of the found links, we start by moving along the link of interest until we find the output widget. This is done by using the circle intersection algorithm again. From the previously found intersections, we create a circle with a very small radius (we use a radius five times smaller than that of the widgets) and with its center where the previous intersection was. We then apply the circle intersection algorithm to find new intersection points. Since the links are created using splines, they will always have smooth turns and will therefore never have sudden changes in direction. This is exploited because in case there was an overlap of different links, we make sure that we stay on the right one by comparing the direction of the newly found points with the previous ones and picking the point for which the direction changes the least. For the first iteration, since there was no previous direction, we consider a direction of 180, since the link should go to the left of the receiving widget. These operations are done iteratively until we get close enough to the right of a widget. Once this happens, we assume that there is a link between the two widgets, and the resulting link position is stored in order to make sure that we do not

follow the same link again. These operations are then repeated for all connected components until all are checked. The pseudocode for this algorithm is given as algorithm 4.

However, the link detection algorithm described above is not perfect. This is evidenced by the fact that if there were any discontinuity in the links, they would not be detected (see figure 3.6.a), or if a link passed through a widget (see figure 3.6.b), then the links would be incorrectly detected. No solution was found for these special cases, but they are rare enough that they do not affect the results of our experiments.



(a) Workflow with link discontinuity



(b) Workflow with link going through a widget

Figure 3.6: Workflows that cause mistakes in the link detection algorithm

Once the widgets and links are recognized, the characterization of the workflow is complete, since we now have all the information we need to uniquely describe the workflow. If there are multiple instances of the same widget, they are distinguished by a number so that we can keep track of which instance we are referring to.

To evaluate the performance of these steps two reference datasets were created with correct information about the widgets and the links, respectively, for the workflows coming from the lecture notes. This way a comparison between the detected widgets and links from the algorithm and the actual widgets and links from the workflows can be performed. The number of workflows, widgets and links identified correctly is used as the evaluation metrics.

Algorithm 4 Link detection

Import the array created by the ”Widget location array creation” algorithm
Get the the detected widget size from the ”Get widget size from image” algorithm
Find which widgets are detected in the screenshot through the widgets array
Initialize with zeros the link array and load the screenshot
Binarize the screenshot through thresholding in order to highlight the links between widgets
Identify connected components in the obtained binary image and label them with integers
for Each identified widget **do**
  Find which labels are in the near vicinity of the identified widgets
  Create two arrays to keep track of which labels are identified to the right of the widgets and which to the left respectively
end for
for Each identified widget **do**
  Extract the connected components labels that are identified as incoming
  for Each of the found labels **do**
    Check if they are also identified as outgoing relative to the other widgets
    if There is exactly one incoming link **then**
      Assert which widget has the outgoing link
      Save in the link array that there is a link between the two widgets
      break
    else if There is more than one outgoing link **then**
      Create a binary image containing just the link of interest
      Use the ”Find Circle Intersection” algorithm to find the link starting points
      for Each found starting point **do**
        $previous\ direction \leftarrow 180$
        $center \leftarrow (x,y)$ of the starting point
        while True **do**
          Use *find_circle_intersection* to find the new center and the new direction
          if $abs(abs(previous\ direction - new\ direction) - 180) < 20$ **then**
            $center = (x_{center} + 1, y_{center})$
          else
            $center \leftarrow new\ center$
            $previous\ direction \leftarrow new\ direction$
          end if
          if The center of the circle is very close to the right of another widget **then**
            Assert which widget has the outgoing link
            Save in the link array that there is a link between the two widgets
            break
          end if
        end while
      end for
      break
    end if
  end for
end for

3.2 *t-SNE* Workflows Representation

Once the workflows are extracted from the screenshots, we need to create a dataset to easily access the data about these workflows. There are over 700 screenshots available in the lectures, but over half of them do not contain a workflow. Since we are only interested in the workflows, the dataset does not need to include these screenshots, leaving us with a dataset of just over 300 screenshots that contain workflows. Two separate datasets will be created with different characteristics: the first one will be used in Orange to visualize the workflows and their differences on a 2D plane, while the second one will be used to extract the information needed to create the prompt that will be used to ask the LLMs to suggest potential new widgets.

To create both datasets, each workflow is embedded with a custom method. A vector is created for each workflow. Each column of the vector contains information about the presence of a particular widget or link in the workflow. If a widget is present in the workflow we are considering, then there will be one in the corresponding column and the same goes for the links. However, as mentioned before, the two datasets are different. The dataset intended for prompt generation takes into account how many instances of a widget are present in the workflow and therefore reports the number of times each widget is present in the datasets. It also takes into account all widgets, whether they are ever used or not, while it only takes into account used links (because if it took into account all possible widgets, the vector would be over 51,000 features long).

The Orange dataset, on the other hand, does not consider multiple instances of the same widget and reports a 1 if it appears in the workflow, regardless of how many times it is present. Furthermore, this dataset only reports information about the widgets that are considered meaningful. A hypergeometric distribution analysis is performed on the widgets used in different chapters of the lecture notes, and if the widget's corresponding probability coming from the hypergeometric distribution is lower than a threshold, then it is considered enriched and therefore meaningful. Furthermore, in both datasets, the widgets *Data/File*, *Data/Datasets*, and *Data/Paint Data* are considered as one, meaning that both their presence is united into the same feature and their links are, too, united. This is because these widgets most of the timess serve the same purpose in the workflows and are basically interchangeable, meaning that a difference in the presence of these three widgets would not provide any valuable information about how similar two workflows are.

After the two datasets are created, the Orange dataset is used to visualize and analyze the different clusters created to highlight the similarities and differences between the different types of analysis performed in each workflow. The figure 3.7 shows the Orange workflow used to

perform the above operations. The other dataset is instead used to find which workflows from the lectures are the most similar to workflow we are considering in the new widget suggestion task. The most similar workflows are then used to retrieve possible next widgets that are then given to LLMs to choose from.

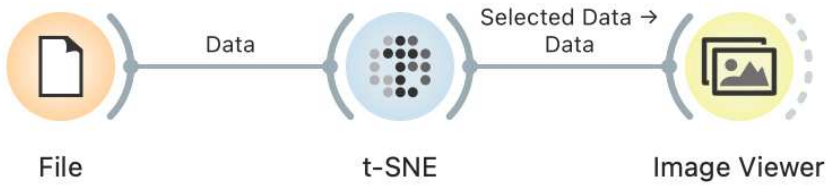


Figure 3.7: Workflow to visualize t-SNE of the workflows dataset

3.3 Initialization of the LLMs to perform the tasks

The LLMs used to perform the tasks and for which the performance is evaluated are GPT-3.5-turbo-0125, GPT-4o, Gemma2 (27 billion parameters), and Llama3 (70 billion parameters). For the first two the OpenAI API was used to retrieve the response for each prompt, while for the latter two, online GPU Slurm cluster FRIDA was used to run the LLMs via Ollama WebUI. The GPT models require payment for each token used both in the input and in the output, while the other two models are open source, therefore, if powerful enough hardware is available, they can be run for free. The GPT and Gemma2 models have a comparable response time, while Llama3's is longer. Every LLM had the same parameter settings:

- $temperature = 0.5$
- $top\_p = 0.3$

In order to make the LLMs perform the tasks we want them to do on a given workflow, we first have to find a way to describe how the latter is built and what the elements in it actually do. This means that all three prompts for the three tasks share two common traits: description of the workflow structure and description of what each widget in the workflow does. To provide an efficient description of the workflow to the LLMs, several methods have been experimented with. As mentioned in the previous chapter, the links between widgets are necessary and sufficient to uniquely describe the workflows, but the order in which they are reported seems to be the most important factor in making the LLMs actually "understand" how the workflow is constructed. Therefore, to give the workflow characterization to the LLMs, we first perform a topological sort on the workflow to find the order of the nodes. Once this order is found, we iteratively report the links belonging to the nodes that come first in the workflow. Therefore, firstly

we start by reporting the links coming from the first node, then those coming from the second, then those coming from the third, and so on, until all the arcs of the DAG are given. The links are reported to the LLMs using the metalanguage creation pattern with the following outline: *"Module1/Widget1 -> Module2/Widget2"*, where *"Module1/Widget1"* is the outputting widget and the *"Module2/Widget2"* is the receiving widget.

For each widget present in the workflow, a description of the widget's function is also given. This, as well as information about what input and output data is available for each widget, was found to be very important to help the LLMs understand the function of a given workflow. The inputs and outputs are also very important because they allow the LLMs to detect possible nuances of what each widget might perform in the workflow, which, as we have reported, may vary from case to case. They are also particularly important for the new widget suggestion, because it limits the range of widgets that are consistent with the outputs available in the workflow we are considering.

For every prompt the same introduction was given at the beginning in order to give background on what we are trying to do and on how the workflow will be described. The introduction uses the persona pattern, the metalanguage creation, and the template pattern techniques and is the following:

"*You are an expert in explaining Orange data mining workflows. Widgets are the working units of Orange and the connections between them allow the user to build workflows able to achieve different goals and perform different tasks.*
I will give you information that allows you to understand how the workflow is built. This information will have the following outline and here the capitalized words work as placeholders for the information about the workflow:

Links in the workflow:
OUTPUTTING WIDGET -> RECEIVING WIDGET
OUTPUTTING WIDGET -> RECEIVING WIDGET

Widget descriptions:
WIDGET NAME: DESCRIPTION OF THE WIDGET
Inputs:
- INPUTS
- INPUTS
Outputs:
- OUTPUT

- OUTPUT

WIDGET NAME: DESCRIPTION OF THE WIDGET
Inputs:
- INPUTS
- INPUTS
Outputs:
- OUTPUT
- OUTPUT

WIDGET NAME: DESCRIPTION OF THE WIDGET
Inputs:
- INPUTS
- INPUTS
Outputs:
- OUTPUT
- OUTPUT"
"

3.4 Workflow name and description generation

The first two tasks we set out to accomplish was to get the LLMs to create a name and a functionality description for a given workflow. This was done through the implementation of a few-shot prompting technique. Five sample workflows were chosen and for each of these a name and two descriptions were created, one that explains what the workflow achieves mostly from an operational point of view, and another that also explains the widgets used and the connections between them. The workflows were specifically selected to be as different as possible between each other in order to give the LLMs a range of examples coming from different workflows functions and structure. The five workflows in question are reported in the Appendix (see figure A.2). Alongside the introduction reported in the previous section, a specific prompt was created for both tasks and are both used right after the introduction. The prompt for name generation is the following:

"I will now give you 5 examples of possible workflow-name pairs. Then I will give you a new workflow. Similarly to what I did in the examples, generate a maximum 5-word-long name for the given workflow that summarizes what it achieves."

The prompt for description generation is the following:

"I will now give you 5 examples of possible workflow-description pairs. Then I will give you information about a new workflow. Try to describe in a short paragraph, similarly to what I did in the examples, what the workflow's task is and how that is achieved through the widgets."

The prompt will then include the characterization of the five sample workflows and the respective output examples for the chosen task. Finally the characterization of the workflow for which we are trying to get the name or description is given and the prompt is then concluded with the line *"Name:"* or *"Description:"*, depending on the task we are trying to perform, to complete the few-shot prompt so that the LLMs only output the name or description chosen for the workflow.

The results coming from both the name generation and the description generation tasks are evaluated on a manually built test set made of ten workflows. This workflows were hand picked to be as varied as possible, in order to consider as many possibilities as possible (see figure A.3). For each workflow a reference name and two reference descriptions were produced. These are then used to compare the output coming from the LLM to the human produced output. A score from 0 to 10 is given to the LLM generation task based on the similarity with the human produced output. In order to automatize the scoring mechanism, a prompt was generated to compare two texts based on similarity and the LLM *GPT 3.5-turbo-0125* was chosen as a baseline for comparing the outputs. This way the comparison should be consistent as there is much lower variability within the same LLM. Since the test set workflows are ten, the sum of the scores produce a percentage score of how efficient each LLM was at performing the task and this will be used to compare the performance of the different LLMs.

The prompt used to get *GPT 3.5-turbo-0125* to score the similarities between two different texts is the following:

"You are an expert in text analysis and comparison. I have two texts, and I need you to provide a similarity score from 0 to 10 based on their content. A score of 0 means the texts are completely different in content, while a score of 10 means they are identical in content. Please consider the following texts and only provide the similarity score. Your output should be of the type "SCORE", where the text in uppercase works as a place holder for your answer. Here are the texts.
Here are some examples:

***Input**:*
'Text 1:

”The cat sat on the mat. It was a sunny day.”

Text 2:
”A dog laid on the rug. The weather was bright and clear.”

Score:’

***Output**:*
’7’

***Input**:*
’Text 1:
”The team lost, because Brian played poorly.”

Text 2:
”The team won, although Brian played poorly.”

Score:’

***Output**:*
’5’

***Input**:*
’Text 1:
”The exam was so easy that every student got an A on it.”

Text 2:
”The exam was extremely hard and no one passed it.”

Score:’

***Output**:*
’0’

Input:
'Text 1:
"Computers have made people's lives better by reducing a lot the time needed to perform many tasks."

Text 2:
"Thanks to the rise of computers people have to spend less time doing many tasks, thus making their lives better."

Score:'

Output:
'10' "

3.5 New widget suggestions for a workflow

The third and final task we tried to achieve was to get the LLMs to suggest widgets that might be added next to a workflow to fulfill a previously selected purpose. Since the number of widgets available is too big to be given to the LLM to chose from, we needed to find a way to lower the number of possible widgets to a more reasonable number, this way both the number of tokens and accuracy of the response would improve. To do this the dataset of the workflows coming from the lectures was used. Given a workflow, it's reasonable to think that if there is another workflow that is similar to it and the latter has more widgets than the one we are considering, then the widgets that are present in the second workflow but not in the first one are probably good candidates for potential new widgets in the original workflow. The more similar the two workflows are, the better the candidates are. Remembering that earlier we defined a way to turn any workflow into a vector, to evaluate the similarity between two given workflows three different methods were used and the respective efficiency were compared:

1. Euclidean distance: L2-norm of the difference between two vectors.
2. Cosine similarity: dot product between two vectors divided by the product of their length.
3. Custom method: when comparing two workflows, the widgets that are present in the incomplete workflow but are not present in the reference are given a weight of 1, while the widgets that are present in the reference workflow but are not present in the incomplete one are given a weight of 0.5. The custom "distance" is calculated by summing up the

weights. This method was implemented because if the incomplete workflow has widgets that are not present in the reference workflow, then the potential candidates coming from the the latter should not be as valuable, because the original workflow might be doing something slightly different. The opposite is usually not true, because this task is being used when the workflow is incomplete and the absence of one or two widgets is exactly what we are looking for.

In all three cases, the lower the defined score is, the more similar the two workflows are. We are now able to compare the vector for the workflow we are considering with the ones coming from the previously built dataset. The ten most similar workflows are then analyzed. The widgets that are present in these workflows, but are not present in the original one are then considered. The list of found possible widgets is then further elaborated. The target length chosen for the list of possible widgets is twenty. Therefore, if the list is shorter than that, the list is augmented. To do that, the embedding of the descriptions of every available widget was performed using the Bert model. This model allows to convert textual data in a vector, thus enabling potential mathematical comparison between different texts [22]. These so found vectors are then used to find the widgets that best relate to the goal the new widget should accomplish using cosine similarity. The widgets that best fit the purpose selected are then used to complete the list.

Once the list of possible widgets is completed, the prompt can be built. The prompt introduction reported previously is used as the opening to the prompt. After the introduction the characterization about the workflow is provided to the LLMs. Then the list of possible widgets is reported, as well as their description, and their possible inputs and outputs. Finally the following is added to finish the prompt:

"Given the information provided about the workflow and about the goal we want to achieve, choose three widgets that make the most sense to be added next to the workflow from the possible ones I provided. For each chosen widget report the percentage probability that it is right and the reason why you think it fits the workflow.

Your output type has to be that of a yaml file to be read in Python as a dictionary. Use the following scheme (replace the PLACEHOLDERS with your output):

"WIDGET NAME1:
- PERCENTAGE PROBABILITY OF BEING RIGHT
- This widget would suit the workflow because REASON

WIDGET NAME2:

- PERCENTAGE PROBABILITY OF BEING RIGHT
- This widget would suit the workflow because REASON

WIDGET NAME3:
- PERCENTAGE PROBABILITY OF BEING RIGHT
- This widget would suit the workflow because REASON" "

In order to evaluate the efficiency of the possible widget list and the performance of the LLMs at performing this task a dataset made of forty incomplete workflows was created. For each workflow the desired widget and the function the new widget achieves were defined. This information was used to evaluate the accuracy of the possible widgets list by checking if the target widget is present in the list. Furthermore, each LLM is asked to recommend a set of three possible widgets with an explanation for each widget. If the target widget is actually recommended by the LLM, then the workflow is marked as guessed correctly. Then the accuracy out of all the forty test set workflows is used as a metrics to assess and compare the efficiency of the LLMs.

Chapter 4

Results

All the operations reported in the previous chapter were specifically chosen to be as efficient and accurate as possible. We will now see how these operations performed and how the LLMs did at understanding and constructing the data mining workflows.

4.1 Workflow detection

We will now analyze the performance of the workflow detection algorithm. For this purpose, we will first present the accuracy of the screenshots analysis by comparing the identified workflow to the actual ones from the screenshots. This is achieved by dividing the number of screenshots correctly analyzed by the total number of screenshots present in the database, which is 757. We will then present the accuracy of the the detected workflows, by dividing the number of workflows detected correctly by the total number of workflows present in the screenshots, which is 308. Finally, the same logic is applied for calculating the detection accuracies of widgets and links within the workflows. It is important to note, though, that if, for example, a widget was wrongfully detected in a screenshot that detection would be classified as an unwanted detection, which undermines the quality of the detection algorithm, but does not impact the calculated accuracy. The same applies for the link detection algorithm.

The workflow detection algorithm is one of the most convoluted ones since it had to work on a large dataset of screenshot all differing from one another, yet it performs very well, allowing us to analyze 96,04% of screenshots and 91.88% of workflows correctly. Furthermore, as can be seen in the table 4.1, the most accurate process is the one for identifying widgets. This is due to the fact that some screenshots show discontinuity in the links between widgets, either because the screenshot was modified in such a way that it interrupts a link (as previously shown in figure

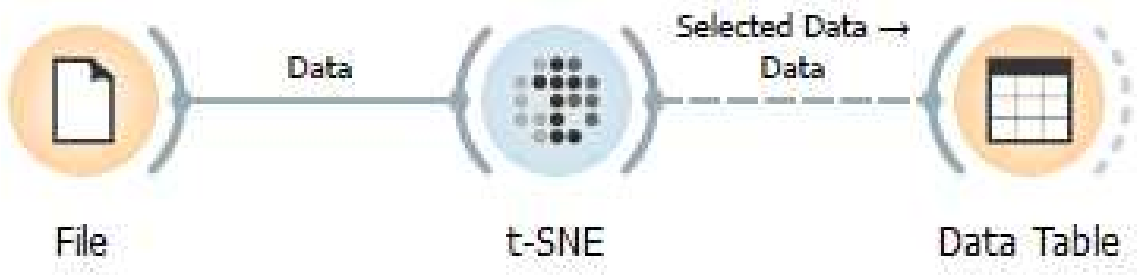


Figure 4.1: Example of link in a workflow drawn as a dashed line

3.6), or because no data instances were selected in the previous widget, thus turning the link in a dashed line (see figure 4.1).

	Detection accuracy	Unwanted detection
Widgets detected correctly	99.10%	4
Links detected correctly	96.05%	2

Table 4.1: Results for the workflow detection algorithms

It is important to mention that when there is a widget that is not recognized in the workflow, this causes the link detection process to be faulty. This is caused by the fact that the positions the link detection algorithm checks are those coming from the widget detection algorithm, therefore a mistake in the latter causes at least one mistake in the former. It is also important to mention that, since these lecture workflows were produced over the years, there are some that present previous versions of widget icons, therefore making the widget detection process ill-posed.

Since the screenshots were not made with the intention for the workflow to be identified, it makes sense that they do not show the best characteristics to allow workflow detection through image analysis, therefore worsening the results. This is also proven by the fact that the workflows specifically created for evaluation and sampling for the LLMs tasks are all identified correctly. All things considered, the detection process works well enough to be considered valid and dependable.

4.2 Workflows distance assessment

Once the dataset of the 308 workflows coming from the lecture notes is created, it becomes possible to visualize their t-SNE representation. As shown in the figure 4.2, the workflows present some clear clusters, but show that there are some mixed and varied ones too. The colors shown in the figure represent the folder from which the workflows are taken, showing that

there are some clusters relative to a specific folder, as well as folders that contain more generic workflows, thus showing mixed colors clusters.

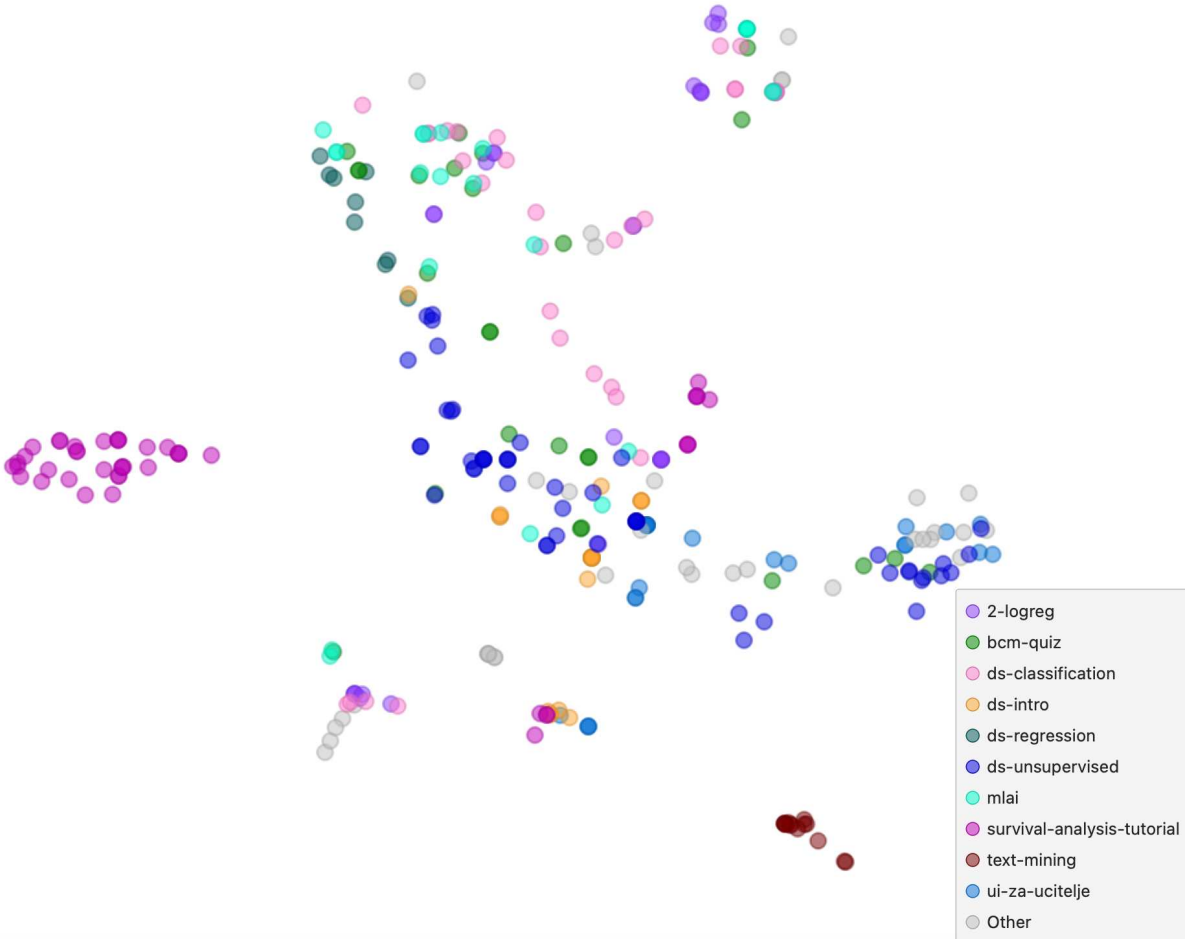
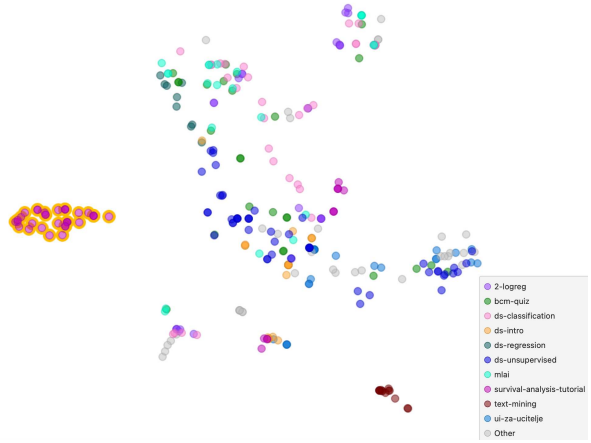


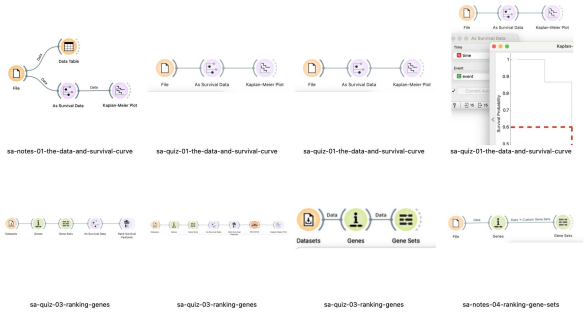
Figure 4.2: t-SNE visualization of lecture workflows

Well defined clusters include for the most part different types of analysis. The purple colored cluster on the left, highlighted in figure 4.3.a, for instance contains only workflows that perform survival analysis. It makes sense that it is far away from everything else, because in order to perform that type of analysis it needs to use widgets that were created specifically to do just that and are not useful in performing any other types of analysis (see figure 4.3.b). Following this logic, it would make sense to assume that the central clusters would contain widgets that are common to almost any workflow. This is in fact true, as shown in figures 4.3.c and 4.3.d, since the workflows belonging to this cluster mostly contain dataset importing widgets, such as *Datasets* and *File*, and visualization widgets, such as *Data Table* and *Scatter Plot*. Finally, it is important to make a consideration even on wide spread clusters, such as the one highlighted in figure 4.3.e: even when we view the workflows (figure 4.3.f) from a cluster as wide spread as this, these still are quite similar to each other and perform basically the same type of analysis

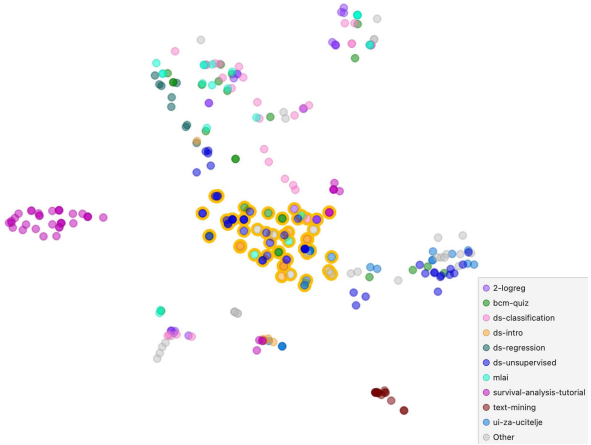
between each other. This allows us to say that the method of analyzing the distance between two workflows to evaluate the similarity between them is reasonable and should allow us to find valuable possible widget lists to give to LLMs to perform the new widget suggestion task.



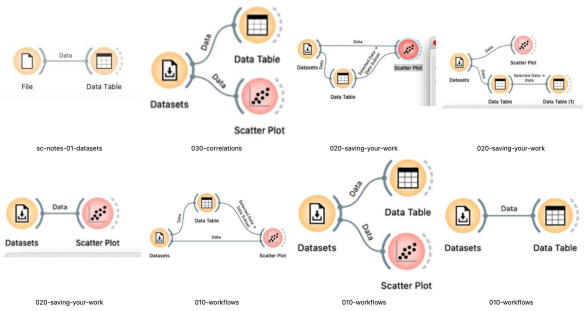
(a) Selection of a clearly defined cluster



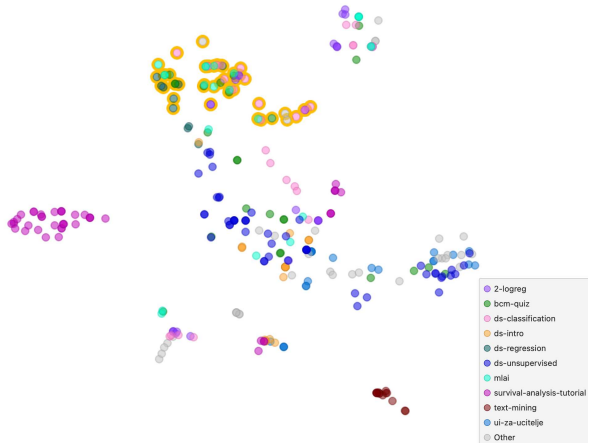
(b) Workflows from the clearly defined cluster



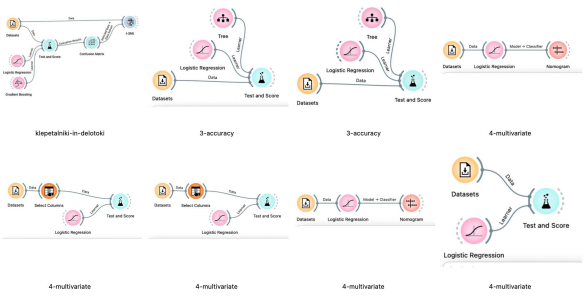
(c) Selection of the central cluster



(d) Workflows from the central cluster



(e) Selection of a wide spread cluster



(f) Workflows from the wide spread cluster

Figure 4.3: Visualization of clusters and workflows within them

Since different methods to establish the distance between two workflows were defined in Section 3.5, it is important to recognize which one performs the best. For each method reported two tests were made: first test only considered information about the widgets present in the workflows, while the second test also considered information about the links. The performance of these methods is then assessed by testing them on the New Widget Evaluation dataset, which is composed by forty incomplete workflows. The list of possible widgets that might be added for these workflows is built as described in Section 3.5, using each of the distance calculation methods defined. The percentage of workflows for which at least one of the possible widgets is correctly identified is used to evaluate the performance of the method used. As shown in table 4.2 the custom method when considering only the widgets performs the best and has an accuracy of 85%. The cosine similarity method also performs quite well in both tests having a similar performance to the custom method when the links are considered too, and has a slightly lower accuracy than the custom method when only the widgets are considered. The Euclidean method performs the worst in both cases and is not accurate enough to be used to create possible widget lists. Given these results, the custom method considering only the widget information is chosen to create the list of possible widgets. This process, however, is far from perfect and causes the performance of the LLMs on the new widget suggestion task to worsen. This is caused by the fact that the list of possible widgets does not include the right widget in 15% of workflows, thus making the LLMs answer an ill-posed question, for which they are not given the right answer to. The reason for this could be that the dataset of workflows coming from the lecture notes is not broad and various enough to be used for any workflow. If the workflows we are considering are similar to the ones present in the dataset, then this method will perform well, otherwise the quality of the possible widgets list is not gonna be as good. This is further proven by the fact that the test set workflows include some types of analysis that are not well covered in the lecture notes, thus making the process of finding possible new widgets more difficult. However, since an accuracy of 85% is still quite high, the LLMs task of suggesting a new widget can still be analyzed and evaluated, while always keeping in mind that the maximum accuracy the LLMs can possibly achieve is 85%.

Percentage of workflows with good possible widget list			
Information considered	Euclidean distance	Cosine similarity	Custom method
Only widgets	72.5%	80.0%	85.0%
Widgets and links	65.0%	82.5%	82.5%

Table 4.2: Performances of the different methods used for workflow similarity comparison

4.3 Workflow name generation

The first LLMs task we will consider is the one for name generation for a given workflow. For both this and the description generation task, as described in Section 3.4, the performance is evaluated on a test set made up of ten workflows. As shown in table 4.3, the performance for this task is quite poor. This is in a way devious, though, because the generation of a 5-word-long name is highly subjective and there is no such thing as a right or wrong answer. While it is true that the names given as output from the LLMs are not really similar to the manually created ones, it is also true that the most of the names the LLMs proposed are still valid and fit the workflows they were created for. Considering that the goal for this thesis is to create a virtual assistant and that this task would be useful mostly for finding a name to save the workflows by, the name suggested by the LLMs can hint the users with a potential name which could then be modified according to the user's needs.

Similarity score between reference names and LLM output				
	GPT-3.5-turbo-0125	GPT-4o	Gemma2 (27B)	Llama3 (70B)
Score	44%	42%	37%	41%

Table 4.3: Performances of different LLMs on the name generation task

In general all LLMs performed similarly, with GPT-3.5-turbo-0125 performing slightly better than the other three LLMs at creating names similar to the reference ones. While there is no clear winner for this task, it is fair to safe that Gemma2 (27B) performed the worst compared to the other ones. As mentioned before, this does not mean that Gemma2 is the worst at generating names, it simply means that the names generated by this LLM are more different from the references than those generated by the other ones.

4.4 Workflow description generation

For the description generation task, two different types of descriptions were created and generated using the LLMs. This was done because just as names can be subjective and change between different interpretations, descriptions can also be subjective. Therefore, two descriptions were manually created for each of the sample and test set workflows. The first is free-form, meaning that there is no structure or rule used to create these descriptions, the second instead follows the order of the widgets from the workflow, making the description generation process more reliable. The results shown in table 4.4 prove this, in fact all LLMs performed significantly better on the descriptions created using the structured method.

Similarity score between reference descriptions and LLM output				
Description type	GPT-3.5-turbo-0125	GPT-4o	Gemma2 (27B)	Llama3 (70B)
Free-form	67%	65%	68%	62%
Structured	93%	94%	87%	84%

Table 4.4: Performances of different LLMs on the description generation task

Similarly to what happened with the name generation task this does not mean, however, that the second type of description is always better. These two types of description also differ in content. The first type is in fact more concise and to the point compared to the second one, which is more prolix. This is because the first type focuses more on the type of analysis performed in the workflow, while the second one also describes how the workflow is structured and how the widgets interact with each other. Both of these types of description can be useful in different use cases and might fit different situations and users. The lower performance of the free-form descriptions is not only due to the subjectivity of the workflow interpretation, though, because when the LLMs are given this much freedom in performing this task the risk for hallucinations increases. This happens when the LLMs output something that is not correct or that is only partially correct. This does make sense, since the LLMs are tools that output text based on the probability that one word might be connected to the next, meaning that when we ask the LLMs to follow a structure we are also indirectly giving them information about how to interpret the workflow and the operations performed within it, therefore lowering the risk for hallucinations.

For the free-form descriptions GPT-3.5-turbo-0125 and Gemma2 (27B) perform the best, while for the structured descriptions the two GPT models perform the best. The score however are pretty much comparable between each other, with the only exception of Llama3 (70B) which has the lowest accuracy on both description styles. Out of the four models, Llama3 has, in fact, an higher tendency to hallucinate, thus being less useful in this use case.

4.5 New widgets suggestion

The third and final task is the one for new widgets suggestion. The performance of this task, as described in Section 3.5, is assessed by applying the task on a test set of forty incomplete workflows. This task is the most complicated one for the LLMs. Not only do they have to understand what the current workflow is doing, they also have to understand what operation the user wants to perform given the specified goal, and which widget fits this purpose among the possible ones. This task's difficulty is proven by the results shown in table 4.5 where it is apparent that the newest and most capable models perform well, while the older and weaker GPT-3.5-turbo-0125 performs the worst. The latter's performance is far from optimal and it

should not be considered usable for new widget suggestions. Gemma2 is the second worst performing LLM, yet it performs decently, having an accuracy of 15% higher compared to 3.5-turbo-0125 and only 5% lower than the other two models. Finally GPT-4o and Llama3 perform the best coming in at an accuracy of 72.5%. While these results are not perfect, they are still quite impressive: keeping in mind that the accuracy for the LLMs is capped at 85% because of the faulty possible widget lists, the GPT-4o and Llama3 out of the non faulty workflows got an 85.3% accuracy. Therefore, assuming that the workflows database could be expanded, thus potentially improving the performance of the possible widget list generation, then the efficiency of this task would be quite promising and potentially very helpful to any inexperienced Orange user. It is also important to note that the goals used for the test set workflows are not at all specific and cover a very broad range of possibilities. In our virtual assistant scenario, the user could be prompted with the option to choose from one of these premade goals or to input his own. In ideal conditions, the more specific the goal is, the more likely the LLMs are to suggest an accurate widget for the given goal.

	GPT-3.5-turbo-0125	GPT-4o	Gemma2 (27B)	Llama3 (70B)
Accuracy	52.5%	72.5%	67.5%	72.5%

Table 4.5: Performances of different LLMs on the new widget suggestion task

Chapter 5

Conclusions

This thesis aimed at exploring and evaluating the LLMs' ability to understand and construct data mining workflows. To do that three critical tasks were performed and evaluated: workflow name creation, workflow description generation and widget suggestion based on incomplete workflows. These three tasks were specifically chosen to lay the foundation for a potential virtual assistant for Orange.

Overall all three tasks showed promising results and significant potential. For the name generation task, although the LLMs did not output names similar to the reference ones, they were still able to produce valid workflow names that could potentially work as a quality-of-life improvement feature to make the process of saving workflows faster and easier. The LLMs ability to generate workflow description was also good and the differences of use and form between the two different styles of description allows for versatility in their use. This feature would both help new users understand premade workflows better, as well as well as potentially make the process of sharing workflows between users easier, as a short description of the types of analysis performed by the other user would be available. Lastly, the widget suggestion task performed quite good with the LLMs achieving up to 85% accuracy under optimal conditions. Implementing this feature within Orange would make it much more accessible to beginners by making the process of creating workflows more trivial.

The starting points and the methods I selected are, however, not perfect and there are many areas of improvement. For instance, the selection of the LLM highly influences the results of these tasks, therefore, choosing the right LLM is essential to have valuable results. So far the best performing LLM seems to be GPT-4o, but with time, the LLMs will get more powerful and capable, so the efficiency of these tasks will probably benefit from this. The power of the LLMs to understand Orange Data Mining workflows, specifically, could also be improved by

fine tuning the models on the toolbox documentation. This is already partially achieved by using the few-shot prompt engineering technique, but the models could probably perform even better if comprehensive fine tuning of the LLMs was performed. Furthermore, the improvement of the starting workflows dataset would be essential to improve the generation of the list for potential widgets, thus indirectly improve the new widget suggestions task, as well.

In conclusion, the integration of LLMs into Orange already looks very promising. In their current state, the techniques explored and used in this thesis already work well enough to be used in the application to aid users in navigating data mining techniques. Further improvements would reduce the complexity of data mining tasks for users even more, thus making them feasible for almost anyone. The rapid evolution and improvement of LLMs also suggests that their role in intelligent virtual assistance for specialized domains like data mining will increasingly expand, thus making them a very powerful resource.

Appendix A

GitHub repository

In this appendix we will cover the main functionalities integrated in the Python code used to perform the operations present in this thesis. The codes and files used, as well as more in-depth documentation, are available at `https://github.com/aletive99/orangescreenshots`. When information has to be exported from or imported to Python, *yaml* files are used. This is done in order to maintain the highest amount of information possible for the variables.

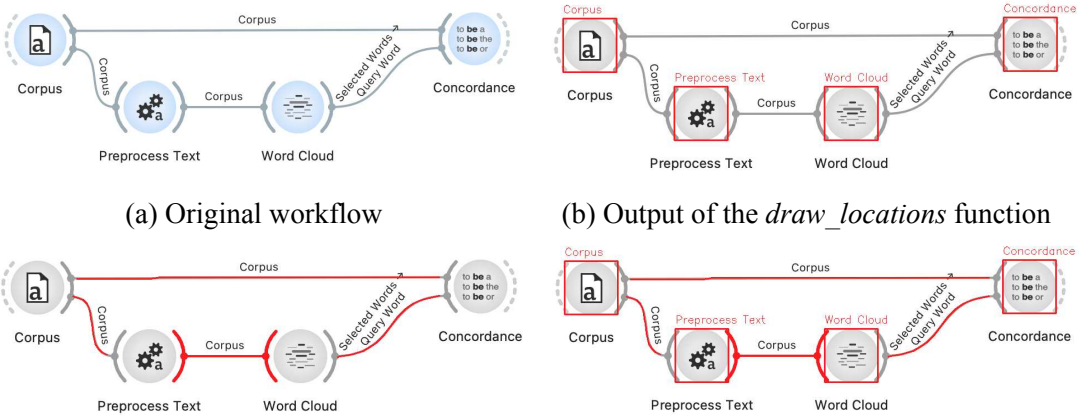
Once the repository is downloaded, the first time the *orangescreenshots.py* library is imported in Python, a download operation of all the available widgets will begin. The widgets are downloaded by parsing the Orange widget catalog available at `https://orangedatamining.com/widget-catalog/` and are then saved in the current working directory inside of a folder named *widgets*. For this library to perform the Orange virtual assistant operations, the user must have a valid OpenAI key saved in their computer's environment.

A.1 Callable functions

The callable functions of the library are the following:

- *draw_locations*: shows the input screenshot with positions of the widgets and their name highlighted in red. Pressing any button closes the image window (see figure A.1.b).
- *draw_links*: shows the input screenshot with identified links highlighted in red. Pressing any buttons closes the image window (see figure A.1.c).
- *draw_links_and_locations*: shows the input screenshot with links, widget name and locations highlighted in red. Pressing any button closes the image window (see figure A.1.d).

- *find_similar_workflows*: returns the ten workflows most similar to the input workflow.
- *get_workflow_name_prompt*: creates and prints a prompt to be used to generate a name of the input workflow.
- *get_workflow_description_prompt*: creates and prints a prompt to be used to generate a description of the input workflow.
- *get_new_widget_prompt*: creates and prints a prompt to be used to get a suggestion for possible new widgets for the input workflow.



(a) Original workflow

(b) Output of the *draw_locations* function

(c) Output of the *draw_links* function

(d) Output of the *draw_links_and_locations* function

Figure A.1: Example usages for the functions that highlight the widgets and links in a given screenshot

A.2 Classes

Two classes were created inside the library. This allows for an easier future integration inside of Orange, in case the virtual assistant gets implemented.

Widget: This class is specifically created to define widgets within a workflow. Its configuration allows to uniquely define which module the widget in question belongs to and its main properties. Once a Widget object is defined, the module and name and the description can be obtained through the specific commands reported in the GitHub page.

Workflow: This class is the center of most of the library, being highly integrated with most of the functions available. It is called once a workflow is detected or needs to be defined from scratch, and allows to perform many different actions on the workflow. The functions designed inside this class allow to:

- Obtain the number of links inside the workflow;
- Obtain a structured text representation of the workflow that reports the links in topological order;
- Obtain a list of the widgets inside the workflow;
- Obtain the description of the widgets present in the workflow;
- Remove a widget and its links from the workflow;
- Interrogate OpenAI API to generate a name for the workflow;
- Interrogate OpenAI API to generate a description for the workflow;
- Interrogate OpenAI API to propose three possible widgets for the workflow.

A.3 Other scripts

Alongside the library, other scripts are available within the GitHub repository. These scripts perform different operations and were all used in the production of this thesis.

A unit test file called *orangescreenshots_test.py* was created to assess the functionality of the main functions from the library. This, being run periodically, ensured that there were no regressions inside the code, and if there were any, one could easily track them and fix them.

The script *data_analysis.py* was specifically created to perform the hypergeometric analysis on the widgets from the workflows and highlight, within a specific yaml file, the widgets and links that are considered enriched.

gemma2_evaluation.sh and *llama3_evaluation.sh* are bash files that were used inside the server partition of the University of Ljubljana in order to evaluate the performance of the virtual assistant operations on Gemma 2 and Llama 3 LLMs respectively. To do that, they both launch a job each on the partition which loads the *orangescreenshots.py* library and runs the *gemma2_evaluation.py* and *llama3_evaluation.py* scripts respectively.

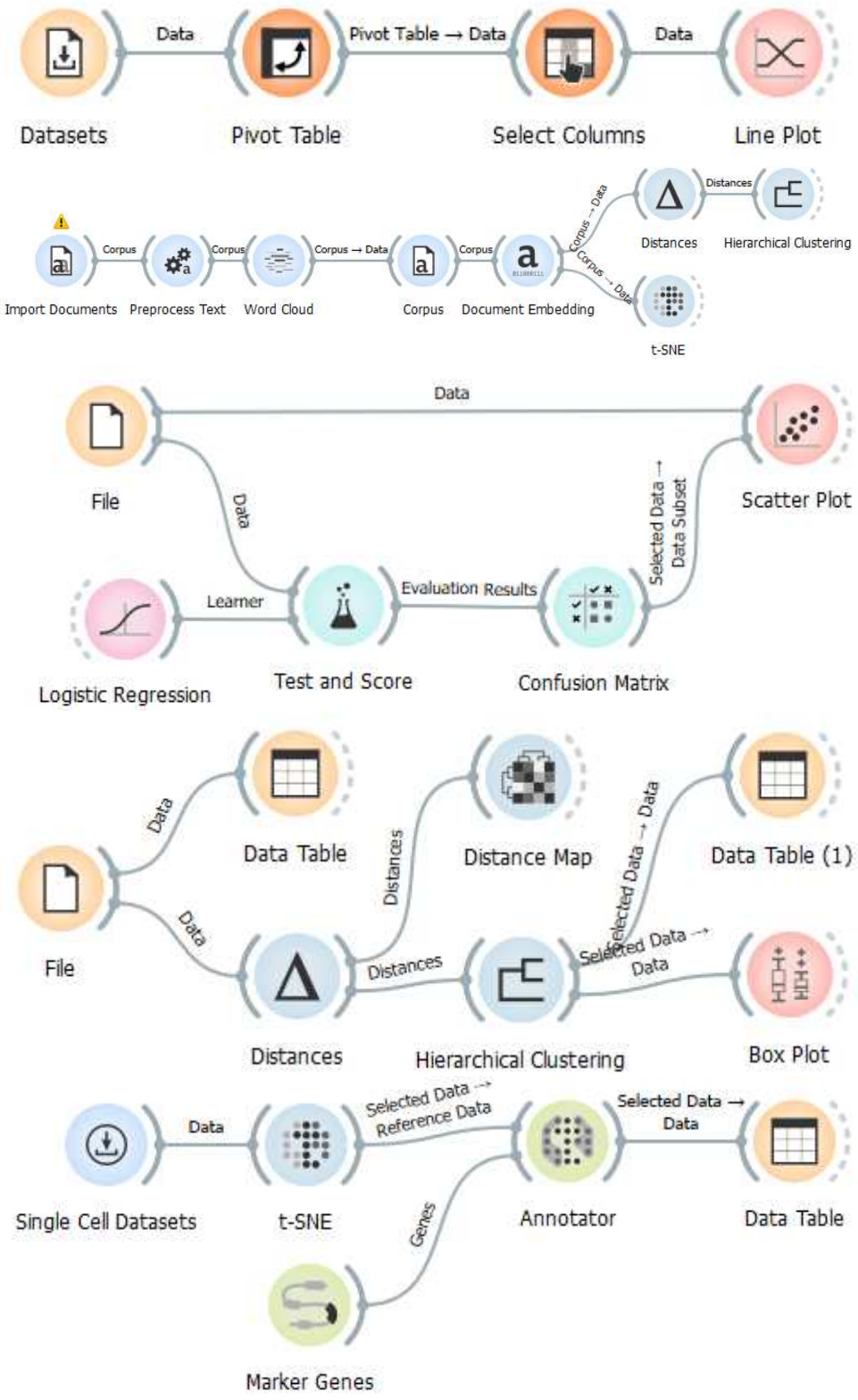


Figure A.2: Workflows used as examples for prompt generation

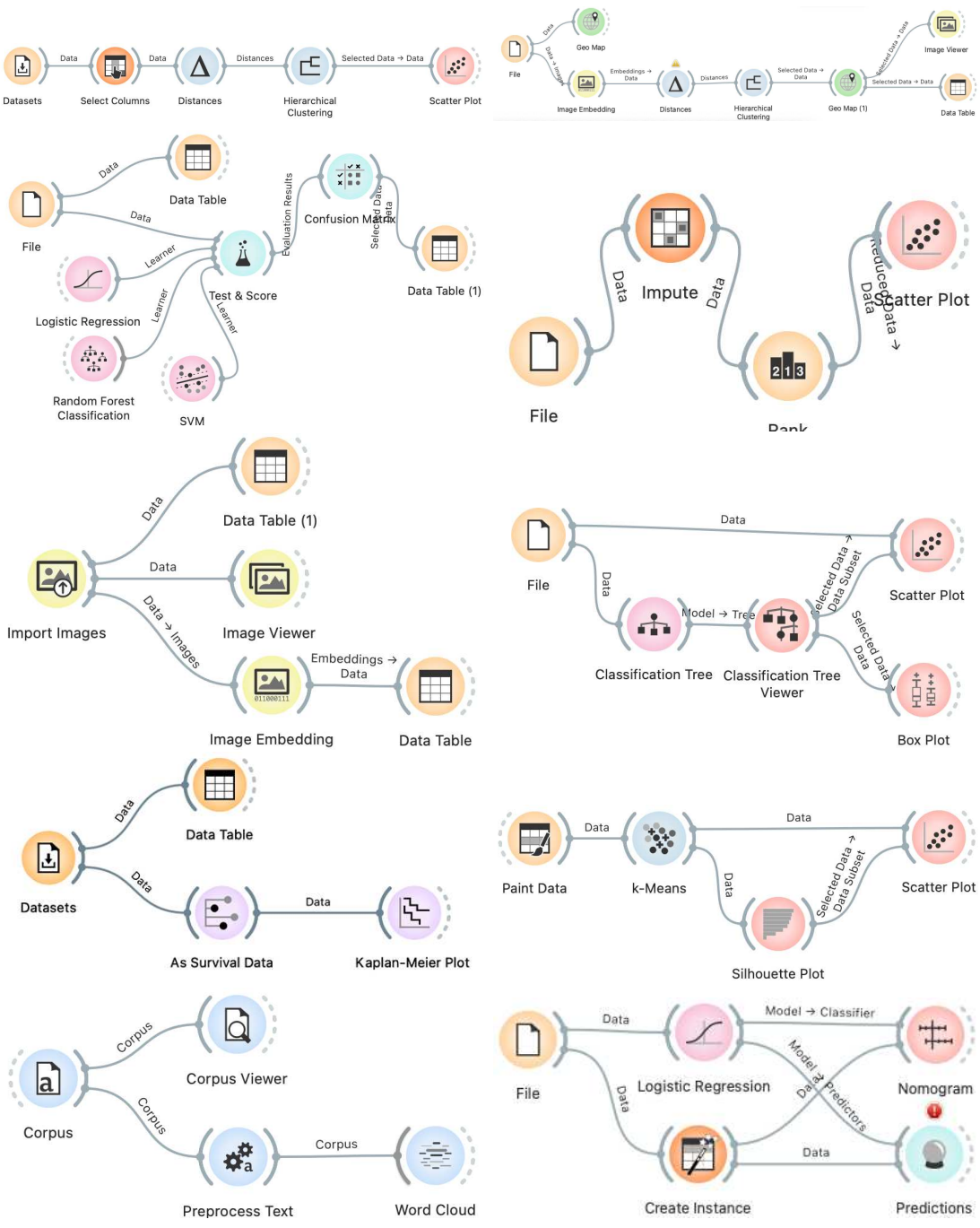


Figure A.3: Test set workflows

Bibliography

[1] Y. Guan, D. Wang, Z. Chu, *et al.*, *Intelligent virtual assistants with llm-based process automation*, 2023.

[2] B. Fatemi, J. Halcrow, and B. Perozzi, "Talk like a graph: Encoding graphs for large language models," in *The Twelfth International Conference on Learning Representations*, 2024.

[3] Q. Wu, Z. Chen, W. Corcoran, M. Sra, and A. K. Singh, *Grapheval2000: Benchmarking and improving large language models on graph datasets*, 2024.

[4] J. Demsar, T Curk, A. Erjavec, *et al.*, "Orange: Data mining toolbok in python," *Journal of Machine Learning Research 14, pgs. 2349–2353*, 2013.

[5] J. Demsar and B. Zupan, "Orange: Data mining fruitful and fun - a historical perspective," *Informatica 37, pgs. 55–60*, 2013.

[6] R. J. Whilson, *Introduction to Graph Theory 4th Edition*. Addison Wesley Longman Limited, 1996.

[7] C.-Y. R. Huang, C.-Y. Lai, and K.-T. T. Cheng, "Chapter 4 - fundamentals of algorithms," in *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, Eds., Boston: Morgan Kaufmann, 2009, pp. 173–234, isbn: 978-0-12-374364-0.

[8] *Graphlib — functionality to operate with graph-like structures*, Accessed: 2024-4-1. [Online]. Available: `https://docs.python.org/3/library/graphlib.html`.

[9] O. Djekoune, "A new modified hough transform method for circle detection," Sep. 2013.

[10] R. Szeliski, *Computer Vision: Algorithms and Applications 2nd Edition*. Springer, 2022.

[11] J. Sarvaiya, S. Patnaik, and S. Bombaywala, "Image registration by template matching using normalized cross-correlation," in *2009 International Conference on Advances in Computing, Control, and Telecommunication Technologies*, 2009, pp. 819–822.

[12] *Opencv: Hough circle transform*, Accessed: 2024-07-23. [Online]. Available: `https://docs.opencv.org/4.x/df/dfb/group__imgproc__object.html#gga3a7850640f1fe1f58fe91a2d7583695dac6677e2af5e0fae82cc5339bfaef5038`.

[13] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2009.

[14] L. F. Khilyuk, G. V. Chilingar, and H. H. Rieke, "Chapter 10 - probability distributions: Discrete case," in *Probability in Petroleum and Environmental Engineering*, L. F. Khilyuk, G. V. Chilingar, and H. H. Rieke, Eds., Gulf Publishing Company, 2005, pp. 145–162, isbn: 978-0-9765113-0-4.

[15] H. Toner. "What are generative ai, large language models, and foundation models?" (2023), (visited on 05/22/2024).

[16] P. P. Ray, "Chatgpt: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 121–154, 2023.

[17] D. Khurana, A. Koli, K. Khatter, and S. Singh, "Natural language processing: State of the art, current trends and challenges," *Multimedia Tools and Applications*, vol. 82, no. 3, pp. 3713–3744, 2023.

[18] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, *A systematic survey of prompt engineering in large language models: Techniques and applications*, 2024.

[19] J. White, Q. Fu, S. Hays, *et al.*, *A prompt pattern catalog to enhance prompt engineering with chatgpt*, 2023.

[20] *Orange data mining - widget catalog*, Accessed: 2024-07-01. [Online]. Available: `https://orangedatamining.com//widget-catalog/`.

[21] *Opencv: Hough circle transform*, Accessed: 2024-07-10. [Online]. Available: `https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html`.

[22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2019. arXiv: `1810.04805 [cs.CL]`.